Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum

JIACHI CHEN, Monash University

XIN XIA, Monash University

DAVID LO, the School of Information Systems, Singapore Management University

JOHN GRUNDY, Monash University

The *selfdestruct* function is provided by Ethereum smart contracts to destroy a contract on the blockchain system. However, it is a double-edged sword for developers. On the one hand, using *selfdestruct* function enables developers to remove smart contracts (SC) from Ethereum and transfers Ethers when emergency situations happen, e.g. being attacked. On the other hand, this function can increase the complexity for the development and open an attack vector for attackers. To better understand the reasons why SC developers include or exclude the *selfdestruct* function in their contracts, we conducted an online survey to collect feedback from them and summarize the key reasons. Their feedback shows that 66.67% of the developers will deploy an updated contract to the Ethereum after destructing the old contract. According to this information, we propose a method to find the self-destructed contracts (also called predecessor contracts) and their updated version (successor contracts) by computing the code similarity. By analyzing the difference between the predecessor contracts and their successor contracts, we found five reasons that led to the death of the contracts; two of them (i.e., *Unmatched ERC20 Token and Limits of Permission*) might affect the life span of contracts. We developed a tool named LIFESCOPE to detect these problems. LIFESCOPE reports 0 false positives or negatives in detecting *Unmatched ERC20 Token*. In terms of *Limits of Permission*, LIFESCOPE achieves 77.89% of F-measure and 0.8673 of AUC in average. According to the feedback of developers who exclude *selfdestruct* functions, we propose suggestions to help developers use *selfdestruct* functions in Ethereum smart contracts better.

Additional Key Words and Phrases: Smart Contract, Ethereum, Selfdestruct Function, Empirical Study

1 INTRODUCTION

The great success of Bitcoin [54] shows the enormous potential of blockchain technology [12]. People usually regard Bitcoin as a representative of blockchain 1.0 [51], the first generation of blockchain technology. In blockchain 1.0, the blockchain technology is usually used to make cryptocurrency [13], e.g., Bitcoin, Ripple Coin [6]. The usage scenario of cryptocurrencies in blockchain 1.0 is limited, as the main application for them is storing and transferring values [35].

The birth of Ethereum [14] changed this situation at the end of 2015. Ethereum leverages a technology named *smart contracts*, which can be regarded as a program that runs on the blockchain.

Authors' addresses: Jiachi Chen, Monash University, jiachi.chen@monash.edu; Xin Xia, Monash University, xin.xia@acm.org; David Lo, the School of Information Systems, Singapore Management University, davidlo@smu.edu.sg; John Grundy, Monash University, John.Grundy@monash.edu.

Smart contracts are usually developed in a high-level programming language, e.g., Solidity [18]. The blockchain technology provides an immutability feature for smart contracts, which means all of the smart contracts are self-executed and can't be modified. Even the creator of the contract cannot modify the code after deploying the contract to the Ethereum. By utilizing a smart contract, developers can easily design their DApps (decentralized applications) [2]. The appearance of Ethereum marked the point that blockchain technology upgraded from blockchain 1.0 to blockchain 2.0. By Sept. 2019, millions of smart contracts on the Ethereum [16] have been applied to different fields, such as gaming [9] and monetization [3], with many other application domains under exploration.

However, the features of Ethereum also make it easy to be attacked. First, Ethereum is a permission-less network; smart contracts on Ethereum can be executed by everyone, including attackers. Second, all the data stored on the blockchain, transactions, and bytecode of smart contracts are visible to the public, which makes smart contracts become attractive targets for attackers. In 2016, attackers utilized a vulnerability (reentrancy [52]) to attack a smart contract owned by an organization named DAO (Decentralized Autonomous Organization). This attack made the organization lose 3.6 million Ethers[1]. People usually call this attack a DAO attack [1]. Actually, the attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts.

This DAO attack attracted great attention from both academia and industry. Some previous works [4, 40] advice contracts add some mechanisms to stop the contracts or transfer the Ethers when emergency situations happen, e.g., a contract being attacked. In this case, the owners can reduce the impact of financial loss. Solidity provides a novel *selfdestruct* function [18]. By calling this *selfdestruct* function, a smart contract can be removed from the blockchain and all the Ethers on the contract will be transferred to a specified address, which is an unique identification for the account[2] on Ethereum.

This *selfdestruct* function is however a double-edged sword for developers. On the one hand, the function enables contract owners have the ability to reduce financial loss when emergency situations happen. On the other hand, this function is also harmful. The function might open an attack vector for attackers. It may also lead to a trust concern from the contract users, as the contract owners can transfer user's Ethers that are stored on the contract. These conflicting features make the *selfdestruct* function valuable to be investigated. **In this paper, we call a smart contract that has executed the *selfdestruct* function as a 'self-destructed contract'.** To better understand the developers' perspective about this unique Ethereum smart contract *selfdestruct* function, we designed an online survey to collect their opinions, and to help us to answer the following research question:

 RQ1: Why do smart contract developers include or exclude selfdestruct functions in their contracts?

We sent our survey to 996 smart contract developers and received 88 responses. Their feedback shows that there are six reasons why developers exclude the *selfdestruct* function. The top two most popular reasons are *security concerns* and *trust concerns*. Developers are worried that the *selfdestruct* function in their contract will open an attack vector for attackers. Besides, this function can also reduce users' confidence of the contract as the contract owner have ability to transfer users' Ethers that stored on the contract balance. To address these concerns of developers, we provide

---

[1]Ether is the cryptocurrency generated by the Ethereum platform. An Ether worth $1270 on Jan. 2021.
[2]There are two types of accounts on Ethereum, i.e., External Owned Account (EOA) and contract account. EOA is controlled by users. Contract account is controlled by its code.

six suggestions in Section 6, which can help developers to better use the *selfdestruct* function. In terms of why developers include a *selfdestruct* function, our survey feedback shows that two thirds of developers will kill their contracts when security vulnerabilities are found, or if they want to upgrade their smart contract's functionalities. After fixing bugs or upgrading the contracts, they will deploy a new version of the contract. This finding inspired our second research question:

RQ2: Why do smart contracts on Ethereum self-destruct?

We called the *self-destructed* contract as a *'Predecessor'* contract, and its upgraded version as a *'Successor'* contract. By comparing the difference between *Predecessor* contract and *Successor* contract, we can identify the reasons why contract destructed, e.g., security reasons. We propose a method that leverages a clone detection tool (SMARTEMBED [42, 43]) to find *Predecessor* contracts and their *Successor* contracts. Then, we summarize 5 common reasons why contracts destructed by conducting *open card sorting* [61].

As a result, we summarize 5 common self-destruct reasons, detailed in Table 2. Two of them – *Unmatched ERC20 Token* and *Limits of Permission* – might affect the life span of contracts. Therefore, an automatic tool to detect these problems would be helpful to extend the life span of smart contracts. This motivated us to investigate our third research question:

RQ3: How can we detect lifespan-based smart contract problems automatically?

We designed a tool named LIFESCOPE, which can be used to detect *Unmatched ERC20 Token* and *Limits of Permission* problems. For *Unmatched ERC20 Token*, LIFESCOPE uses ASTs (Abstract Syntax Trees) to parse the source code and extract related information. LIFESCOPE obtains 100% of F-measure for detecting this problem. For *Limits of Permission*, LIFESCOPE first transfers code to a TF-IDF representation and utilizes a machine learning method to predict the permission. LIFESCOPE achieves an F-measure and AUC of 77.89% and 0.8673 for this task.

The main contributions of this paper are:

- To the best of our knowledge, this is the most comprehensive empirical work that investigates the *selfdestruct* function of smart contracts in Ethereum. We conduct an online survey to collect feedback from developers. According to this survey feedback, we summarize 6 reasons why developers add *selfdestruct* functions and 6 reasons why they do not add them to their smart contracts.

- We design an approach to find 5 reasons why smart contracts self-destructed. These self-destruct reasons can be used as a guidance when practitioners develop their contracts. Also, our approach gives inspiration for researchers. They can use the same approach to find more self-destruct reasons and apply the method to other smart contract platforms, e.g., Ethereum Classic[3] [10].

- We propose a tool named LIFESCOPE to detect two problems that might shorten the life span of smart contracts. LIFESCOPE obtains 100% of F-measure in detecting *Unmatched ERC20 Token*. And it achieves an F-measure and AUC of 77.89% and 0.8673, respectively in detecting *Limits of Permission*.

- According to the feedback from our survey, there are six common reasons why some developers do not use *selfdestruct* function. We give five suggestions for developers to address these issues and to help them better use the*selfdestruct* function in their smart contacts.

The organization of the rest of this paper is as follows. In Section 2, we present the background knowledge of smart contracts. Then, we show the answer to the three research questions in Section 3-5, respectively. We discuss the implication, how to better utilize *selfdestruct* function and threats

---

[3]Ethereum Classic is another popular blockchain platform which support the running of smart contracts.

to validity in Section 6. After that, we introduce related works in Section 7. In Section 8, we conclude the whole work and present our future work.

2 BACKGROUND

In this section, we briefly introduce the background information about smart contracts, the Ethereum system, and some features and knowledge about smart contract programming.

2.1 Smart Contracts

Bitcoin was the first cryptocurrency that utilized blockchain as its underlying technology. It allows users to encode scripts to process transactions. However, the scripts on Bitcoin are not Turing-complete, which restricts the usage of Bitcoin [54]. In contrast, Ethereum leverages a technology named *smart contracts*. These can be regarded as self-executed programs that run on the blockchain. When developers deploy smart contracts to Ethereum, the source code of the contracts will be compiled into bytecode and reside on the blockchain forever. The storage of Ethereum is very expensive, as all the data stored on the blockchain will be copied on each node, a so-called distributed ledger. To minimize the data space, the source code of the smart contracts will not be stored on the blockchain. Once a contract is deployed to the blockchain, the contract is identified by a 20-byte hexadecimal address. Arbitrary users can call the functions of a smart contract by sending transactions to the contract address.

```solidity
pragma solidity ^0.4.25;
contract Example{
  address owner_addr;
  address[] participators;
  uint participatorID = 0;
  function constructor(){
    owner_addr = msg.sender;
  }
  function () payable{
    if(msg.value != 1 Ether)
      revert();
    participators[participatorID] = msg.sender;
    participatorID++;
    if(this.balance == 10 Ether)
      getWinner();
  }
  function getWinner(){
    uint random = uint(block.blockhash(block.number)) % participants.length;
    participators[random].transfer(9 Ether);
    participatorID = 0;
  }
  modifier onlyOwner{
    if(msg.sender != owner_addr)
      _;
  }
  function Selfdestructs(address addr) onlyOwner(){
    selfdestruct(addr);
  }
}
```

Listing 1. A simple contract

Listing 1 is an example of a smart contract that implements a simple gambling game by using Solidity [18]. Solidity is the most popular smart contract programming language on the Ethereum

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:5

platform. Users can send 1 Ether to the contract. Once the contract receives 10 Ethers, the contract will choose 1 user as the winner randomly and send 9 Ethers to him/her.

The first line is called the version pragma, which is used to identify the compiler version of the contract. Lines 3-5 are the global parameters, and the function on line 6 is the constructor function of the smart contract. The constructor function can only be executed once when deploying the contract to the blockchain. Therefore, this function is usually used to store the owner's information. Specifically, line 7 stores the owner's address by using *msg.sender*. (*msg.sender* is used to obtain the address of the transaction sender.) Function on line 9 named fallback function, which is the only unnamed function of the smart contract. This function will be executed automatically when an error function call happens. For example, a user calls function "δ", but there is no function named "δ" in the contract. In this situation, a fallback function will be executed to handle the error call. If the fallback function is marked by a keyword named *payable*, the fallback function will also be executed automatically when the contract receives Ethers. Lines 10 and 11 guarantee that each user sends 1 Ether to the contract. If the user sends other amounts of Ethers, the transaction will be rolled backed by executing *revert()*, which is a function provided by *Solidity*. When the contract receives 10 Ethers (line 14), the contract will choose 1 user to send 9 Ethers by using function *getWinner* (line 17). The contract generates a random number by using the block info related functions[4] in line 18. Then, the contract sends 9 Ether to the winner in line 19.

2.2 Function Modifier

Ethereum is a permission-less network – everyone can call methods to execute smart contracts. Developers usually add permission checks for permission-sensitive functions. For example, the contract in Listing 1 records the owner's address in its constructor function (line 7). In this case, the contract can compare whether the caller's address is the same as the owner's address. Solidity provides *Function Modifiers* which are used to add prerequisites checks to a function call. A function with function modifier can be executed only if it passes the check of the modifier.

Listing 1 line 22 shows a modifier named *onlyOwner*. This modifier requires the transaction creator (*msg.sender*) should be the owner of the contracts (*owner_addr*). Function *Selfdestructs* on line 26 contains this modifier. Therefore, only the owner of the contract can call *selfdestruct* function in line 27.

2.3 Selfdestruct Function

The *selfdestruct* function in Listing 1 line 27 is the only way to remove the contract from Ethereum. When executing this method, the caller can transfer all Ethers on balance to a specific address (*addr*) (line 27). Then, the contract will be discarded. If others transfer Ethers to the self-destructed contract address, the Ethers will be locked forever. Calling *selfdestruct* function when a contract is no longer needed can help clean up the Ethereum environment. To motivate this, Ethereum refunds up to half of the gas used by a contract transaction calling the *selfdestruct* function to the transaction sender. This mechanism is also utilized by GasToken [15], which allows users to store gas when the gas price is low and use the gas when it is expensive. Specifically, a user can create a simple contract (GasToken) that contains *selfdestruct* function when the gas price is low. Then, the user can destruct the GasToken to save the gas when the gas price is high. However, GasTokens also have the downside to the Ethereum network, as it leads to the creation of millions of "useless" contracts, which is against the original motivation of the gas refund. EIP-3529 [65] and EIP-3298 [64] are two Ethereum improvement proposals that suggest reducing the gas refunds.

---

[4](block.blockhash and block.number are the functions provide by Solidity to obtain block related information. Since block hash number is random; so it can be used to generate random numbers sometimes.)

EIP-3529 recommends reducing the gas refund from up to 1/2 gas used by a transaction to 1/5, and EIP-3298 even recommends removing the gas refund directly. These two EIPs imply that the GasToken might be nullified in the future.

The *selfdestruct* function is sometimes harmful as the immutability feature can be broken. Immutability is a special and important feature of smart contracts compared to traditional programs. Once a contract is deployed to the blockchain, none can modify the contract, even the owner. However, this function can allow the owner to kill the contract and make the contract disappear from the blockchain. This might reduce the confidence of the users, as the owner can transfer all the Ethers of the contract. For example, the owner can transfer all the Ethers by calling the *selfdestruct* function on contract in Listing 1 when the contract receives 9 Ethers. In this case, all the users are losers.

```
1  contract Victim {
2      mapping(address => uint) public userBalannce;
3      function withDraw(){
4          uint amount = userBalannce[msg.sender];
5          if(amount > 0){
6              msg.sender.call.value(amount)();
7              userBalannce[msg.sender] = 0;
8          }
9      }
10         ...
11 }
12 contract Attacker{
13     function() payable{
14         Victim(msg.sender).withDraw();
15     }
16     function reentrancy(address addr){
17         Victim(addr).withDraw();
18     }
19         ...
20 }
```

Listing 2. The Demo of the DAO Attack

2.4 The DAO Attack - A Motivation Example of the *selfdestruct* Function

In 2016, attackers found a vulnerability named Reentrancy [40, 52] in a smart contract of the Decentralized Autonomous Organization (DAO organization), and this vulnerability made the DAO organization lost 3.6 million Ethers ($270/Ether on Feb. 2020). People usually call this infamous attack a DAO attack.

List 2 is a demo of the DAO attack. There are two smart contracts, i.e., *Victim* contract and *Attacker* contract. The *Attacker* contract is used to transfer Ethers from *Victim* contract, and the *Victim* contract can be regarded as a bank, which stores the Ethers of users. Users can withdraw their Ethers by invoking *withDraw()* function. However, *withDraw()* function contains the *Reentrancy* vulnerability in line 6-7.

First, the *Attacker* contract uses *reentrancy()* function (line 16) to invokes *Victim* contract's *withDraw()* function in line 3. The *addr* in line 17 is the address of the *Victim* contract. Normally, the *Victim* contract sends Ethers to the callee in line 6, and resets callee's balance to 0 in line 7. However, Ethereum does not support concurrency, which means *Victim* contract sends Ethers to Attacker contract before resetting the balance to 0. When the *Victim* contract sends Ethers to the *Attacker* contract, the fallback function (line 13) of the *Attacker* contract will be invoked automatically,

and line 7 is not executed at that time. So, the *Attacker* contract can invoke *withDraw()* function repeatably.

Actually, the DAO attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts. If the contract contains a *selfdestruct* function, the DAO organization can transfer all the Ethers easily, and reduce the financial loss.

2.5 ERC20 Standard

Motivated by the great success of Bitcoin, thousands of cryptocurrencies have been created in recent years. However, most of them do not have their own blockchain system. Instead, they are usually implemented by smart contracts that run on the Ethereum, also called *tokens*. To ensure different tokens can interact accurately and be reused by other applications (e.g., wallets and exchange markets), Ethereum provides ways to standardize their behaviors. ERC20 [3] is the most popular token standard on Ethereum. It defines 9 standard interfaces (3 are optional) and 2 standard events. To design ERC20 tokens, developers should strictly follow the standard. For example, the standard method *transfer* is declared as "*function transfer(address _to, uint256 _value) public returns (bool success)*". This function is used for transferring tokens to a specific address (_to). The ERC20 standard requires this function to throw an exception if the caller's account balance does not have enough tokens to spend. Besides, the function should fire an event named "*TRANSFER*" to inform the caller whether the tokens are transferred successfully.

2.6 Card Sorting

Card sorting is a research method to organize data into logical groups [61]. Due to the low-tech and inexpensive nature of card sorting, it is widely used to help users understand how users would organize and structure the data that makes sense to them. The users who conduct a card sorting process first need to identify the key concepts and write them into labeled cards, which can be actual cards or a piece of paper. Then, they are asked to classify them into groups that they think are appropriate. By utilizing card sorting, users can design workflow, architecture, category tree, or folksonomy.

There are three kinds of card sorting, i.e., open card sorting, closed card sorting, and hybrid card sorting. Open card sorting is commonly used for organizing data with no predefined groups. Specifically, each card will be clustered into a group with a certain topic or meaning first. If there is no appropriate group, a new group will be generated. All the groups are low-level subcategories and will be evolved into high-level subcategories further. Closed card sorting is used for organizing data with predefined groups. Each card is required to cluster into one of the groups. Hybrid card sorting combines open card sorting and closed card sorting. Hybrid card sorting has predefined groups but allows the creation of new groups during the process.

3 RQ1: DEVELOPER'S PERSPECTIVE ABOUT SELFDESTRUCT FUNCTION

3.1 Motivation

Usage of the *selfdestruct* function can enable developers to destruct their contracts and transfer Ethers when emergency situations happen, e.g. a contract is being attacked or is found to be buggy. However, this function is also harmful for both contract users and contract owners. In our analysis, we crawled all of the 54,739 verified smart contracts from Etherscan [16] by the time of writing (for details see Section 4.2.1), and found 2,786 (5.1%) smart contracts contain a *selfdestruct* function in their source code. In this RQ, we aim to investigate the developers' perspective about using the *selfdestruct* function in Ethereum smart contracts. By understanding the reason why they include

or exclude *selfdestruct* functions in their contracts, we can better understand the advantages and disadvantages of this function. We then want to design some guidance about using the *selfdestruct* function (can be found at Section 6.2), which enables developers to design a more robust smart contract.

3.2 Approach

3.2.1 Validation Survey. In this paper, we utilize the methods proposed by Kitchenham et al. [49] to design a survey for collecting the opinions from smart contract developers. To increase the response rate, we make the survey anonymous [63] and provided a raffle for developers who take part in our survey. Participation in the raffle is voluntary; we chose two respondents who provided their email addresses as the winner, and gave them $50 Amazon gift cards as the reward. We first use a small scale survey to collect feedback about our survey. The feedback includes: (1). Whether the expression about our question is easy to understand. (2). Whether the time to finish the survey is reasonable. After the small scale survey, we refine our questionnaire based on the feedback we collected. Finally, conduct a large scale investigation to collect our data. [5]

3.2.2 Survey Design. To understand the background of the respondents better, we first collect their demographic information. These five questions can help us have an overall understanding of the respondents.

    a. Demographics:
- 1. Professional smart contract developer? : Yes / No
- 2. Involved in open source software development? : Yes / No
- 3. Main role in developing smart contract: Testing / Development / Management / Other
- 4. Experience in years (decimals ok)?
- 5. Current country of residence ?

After that, the respondents are required to choose yes / no in the question 6. If they choose yes, they are required to answer question 7; otherwise, they should answer question 8. Both question 7 and 8 contain a textbox, which enables respondents to input their answer. (There is no length requirement / restriction of their inputs.)

    b. Questions about selfdestruct Functions:
- 6. Will you add *selfdestruct* functions in your future smart contracts? : Yes (Go to Q7) / No (Go to Q8)
- 7. Why do you add *selfdestruct* functions?
- 8. Why do you not add *selfdestruct* functions?

To increase the response rate, we prepare two kinds of survey[6], i.e., English Version and Chinese Version, as Chinese is the most spoken language and English is an international language in the world. The Chinese version survey is carefully translated to ensure the contents between the two versions are the same.

3.2.3 Recruitment of Respondents. To receive sufficient response from different backgrounds, we first sent our questionnaire to our contacts who are working in world-famous blockchain companies or doing related research in academic institutions, e.g., *Ant Financial, The Hong Kong Polytechnic University, NUS, The University of Manchester*. Then, we also collect developers' email addresses on their Github homepage who are contributing to open-sourced blockchain projects. We collected 1,238 email addresses from Github. Due to the scale of the smart contract projects, 1,238 are the

---

[5]ETHICS COMMITTEE APPROVAL

[6]The two surveys and related feedback can be found at: https://github.com/Jiachi-Chen/Selfdestruct

numbers of contributors of the top 100 most popular (ranked by stars) smart contract related projects, which is a good number compared to previous smart contract related surveys [24–26] .

3.3 Result

Since some email addresses we collected are illegal or abandoned, we successfully sent our survey to 996 developers, and receive 88 responses from 32 countries (The response rate is 8.84%). The top three countries in which respondents reside are China (29.89%), the USA (8.05%) and the UK (5.57%). Three of the respondents claim that they are not professional smart contract developers and have no experience in developing smart contracts. Therefore, we exclude their responses and use the remaining 85 responses for analysis. The average years of experience in developing Ethereum smart contracts are 1.96 years (standard deviation is 1.05) for all of our respondents. As the survey was undertaken in Sept. 2019 (about 4 years since Ethereum was first published), 1.96 average years of experience shows that they have good experience in developing smart contracts. Among these respondents, 62 (72.94%), 10 (11.76%), 5 (5.88%) described their job roles as development, testing, and management, respectively. The other 8 respondents said they have multiple roles, such as security auditor and research.

Guided by previous works [26, 70], we performed open card sorting [61] to analyze the survey feedback to summarize the reasons why developers include or exclude the *selfdestruct* function in their smart contracts. Feedback that we received in Chinese was first translated into English. After that, we manually converted each feedback into several separate units with coherent meaning, as some feedbacks contain several reasons. Then, a card was created for each separate unit with a title (ID) and description (feedback content). Two experienced smart contract researchers were involved in the card sorting. The detailed steps are:

Iteration 1: Two researchers randomly chose 20% of the cards. Each card was analyzed by the two researchers together. They were required to summarize a detailed reason, e.g, Security concern, Trust concern. If the root concern is unclear, they omit the card from the card sort.

Iteration 2: The same two researchers analyzed the remaining 80% of the cards independently by following a similar method as iteration 1. Some new reasons are found in this step. After they have gone through the cards independently, they compared their results and discussed any differences. Finally, 6 reasons for including and 6 reasons for excluding *selfdestruct* function were summarized.

There is a threat that what developers told us, i.e., their reasons for adding a self-destruct function to their smart contracts, may be different from what they do in reality. We do not claim that our user study is final and complete; rather, we view it as a first step to better understand the usage of *selfdestruct* function. We invite others to replicate our study with additional surveys and interviews.

3.3.1 Reasons for including the selfdestruct function. 33 (38.82%) of the respondents claim that they will add the *selfdestruct* function in their smart contracts. We analyzed the feedback of these respondents and summarized five key reasons. **As some respondents give more than one reason, the sum of these is higher than 33.**

Reason 1: Security Concerns. 18 (54.55%) respondents claim that they use the *selfdestruct* function to stop the contracts when security vulnerabilities are detected in their contracts. After fixing the vulnerabilities, they can deploy a new contract.

Reason 2: Clean Up Environment. Blockchain is a distributed ledger where each node stores all the data. After destructing the contracts, the functions of the contracts cannot be called anymore. **11 (33.33%) respondents** mention that when the duty of the contract is finished, they will call the *selfdestruct* function to remove the contracts from the blockchain, which can clean up the blockchain environment.

Reason 3: Quickly Withdraw Ethers. By using the *selfdestruct* function, the owner of the contract can remove all the Ethers to a specific address. **9 (27.27%) respondents** claim this function can help them transfer assets quickly.

Reason 4: Upgrade Contracts. **4 (12.12%) respondents** said they may need to upgrade their contracts in the future. Adding a *selfdestruct* function is the easiest method to upgrade their contract. This function allows them to remove the old version of the contract and deploy a new version.

Reason 5: Business Requirement. The business requirement is also a reason why developers add *selfdestruct* function. **2 (6.06%) respondents** said their business partners require them to add the *selfdestruct* function.

Reason 6: Gas Refund. As we introduced in Section 2.3, the gas refund feature of the *selfdestruct* function allows the transaction sender to get up to half of the gas back. **1 (3.03%) respondent** mentioned that he/she adds the *selfdestruct* function to get the gas back.

According to our survey, 22 / 33 respondents claim that they add *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. Besides, the survey feedback also show that *selfdestruct* function is useful for contract developers to handle emergency situations, e.g., when serious security issues are found in the contracts.

3.3.2 Reasons for excluding the selfdestruct function. 52 (61.18%) of the respondents claim that they will not add *selfdestruct* functions in their smart contracts. **As some respondents give more than one reason, the sum of these is higher than 52.**

Reason 1: Security Concerns. The *selfdestruct* function can also lead to serious security problems if the contract does not handle access permissions correctly or the private keys of owners are leaked. **19 (36.54%) respondents** worried that the *selfdestruct* function might open an attack vector for adversaries to exploit. Limiting the permission of calling selfdestruct function is not difficult. For example, a contract can only allow specific addresses to execute this function. However, it is also possible that the private keys [7] of these addresses might be stolen. Once the private keys are stolen by attackers, the smart contract can then be destructed by attackers and all the Ethers will be lost.

Reason 2: Trust Concerns. **16 (30.77%) respondents** who give this reason believe that including a *selfdestruct* function might lead to trust concerns from the contract users. To be specific, a *selfdestruct* function allows the owner to kill the contract and make the contract disappear from the blockchain. Also, the owner can transfer all the Ethers, which raises a trust concern for the user.

Reason 3: Requirement Concerns. **16 (30.77%) respondents** mention that their contracts do not use *selfdestruct* function as their contracts do not have Ethers. Therefore, they do not need to transfer Ethers. Besides, when they want to add some new functionalities, they said they could deploy a new smart contract and ignore the old one.

Reason 4: Unfamiliarity. **7 (13.46%) respondents** claim that they are unfamiliar with *selfdestruct* function. They are worried that they might misuse the *selfdestruct* function, and lead to the bugs.

Reason 5: Additional Complexity. **4 (7.69%) respondents** told us that they need to add more tests if they add *selfdestruct* functions in the contracts, which can introduce additional complexity to their contracts.

---

[7]There are two kinds of accounts in Ethereum, i.e., externally owned account (EOA) and contract account. A contract account is controlled by its code, and an EOA is controlled by the private key.

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:11

Table 1. Information for the 756 self-destructed contracts

	Max	Min	Avg.	Median
Life Span	879.6 days	≤ 1 hour	40.8 days	4.3 days
No. Trans	100582	2	538.8	7
No. Eths	208.6	0	0.44	0

Reason 6: Additional Financial Risk. Risk of losing Ether after destroying the contract is also a concern for the developers. **2 (3.85%) respondents** worried that people may send Ethers to the self-destructed contract, and these Ethers will be locked forever.

According to our survey feedback, *selfdestruct* function might be risky for both contract developers and contract users. We find six reasons why developers destruct their contracts, and the reasons "Security concern" and "Upgrade contracts" can lead to the redeployment of smart contracts. These two reasons are also the most common reasons (66.67%) why developers destruct their contracts. This finding gives us the motivation of RQ2 that we can find some security issues by comparing two versions of contracts.

4 RQ2: REASONS FOR SELF-DESTRUCT

4.1 Motivation

According to our survey, 22 out of 33 respondents claim that they add a *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. Therefore, by comparing the difference between the two versions of the contract, we can find the reasons why contracts self-destructed. Consider the following scenario.

Bob is a smart contract developer. He developed a smart contract several weeks ago, and his company uses this contract to receive money from other companies. However, they find that a function in the smart contract does not limit the caller's permission, which can lead to serious security problems. Therefore, Bob has to destroy the contract by involving *selfdestruct* function and deploy a new contract to the blockchain. The new contract adds a permission check to avoid this vulnerability. Bob and his colleagues try their best to inform other companies not to transfer Ethers to the self-destructed contract anymore. However, it requires a long time to inform all companies. Many users still transfer Ethers to the self-destructed contract, and all the Ethers send to the contract are lost forever. It causes a great financial loss to Bob's company.

From this scenario, we see that calling the *selfdestruct* function may lead to great financial loss. Therefore, we should try to make contracts robust. If we tell Bob that many previous contracts are destructed because a function in the smart contract does not limit the caller's permission, he might check whether his contract contains the same problem and can avoid this problem.

In this section, we compare the self-destructed contracts and their successor contracts to summarize reasons why contracts self-destructed. The reasons we identify can guide smart contract developers and help them refine their contracts.

4.2 Approach

Figure 1 depicts the detailed steps to identify the reasons why some smart contracts have been destructed. Our method consists of three stages. In the first stage, we crawl all verified contracts and their transactions from Etherscan. We crawled 54, 739 smart contracts altogether. We found that 2,786 (5.1%) of these smart contracts among 54,739 contracts contain a *selfdestruct* function, and 756 (27.14%) contracts have been destructed. In the second stage, we first divide crawled contracts into several groups by their creators' addresses. In this case, we can find smart contracts that are
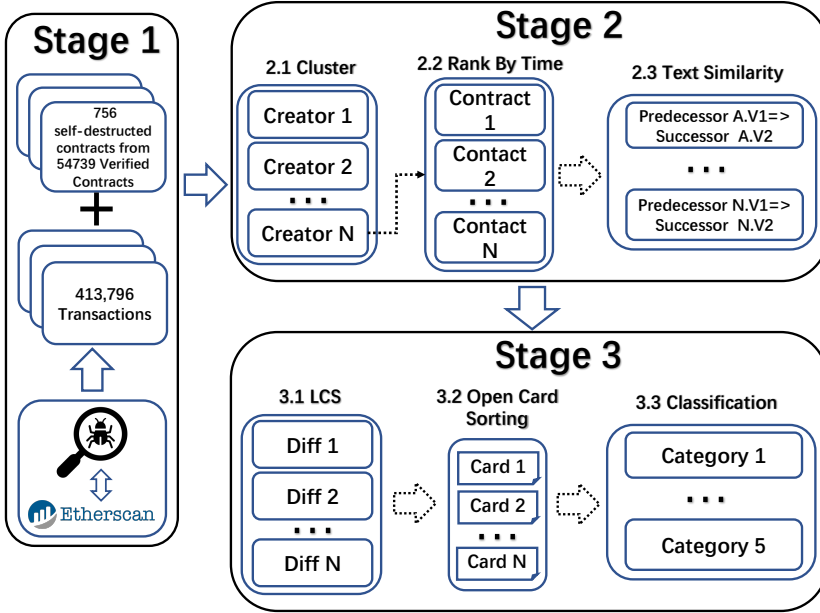
Fig. 1. Overview architecture of finding the self-destructed reasons

created by the same authors. We only use groups that contain self-destructed contracts and rank all the contracts in the same group by contracts' creation time. Then, we compute the code similarity of contracts in each group to find self-destructed contracts (also called predecessor contracts) and their successor contracts. In the last stage, we compare the difference of predecessor contracts and their successor contracts by using *open card sorting*. Finally, we summarized 5 reasons why contracts self-destructed.

4.2.1 Stage 1. Data Collection: Stage 1 is used to collect data for the following two stages. Our data contains three parts, i.e., *verified contracts*, *self-destructed contracts*, and *contract transactions*.

   Verified Contracts: Verified contracts are crawled from Etherscan. To crawl the source code of verified smart contracts from Etherscan, we first need to know the contract addresses of verified contracts. Figure 2 is a smart contract on Etherscan. By obtaining the contract address, we can easily download the source code, transactions, and other information of the contract. The contract address list is provided by Etherscan (https://etherscan.io/contractsVerified). However, Etherscan only shows the last 500 verified contract addresses since Jan. 2019. The data used in this paper are crawled before Jan. 2019 and the lasted 500 verified contracts by the time of the writing. We finally obtained 54, 739 verified contract addresses. After obtaining the contract addresses, we can crawl the source code from Etherscan directly.

   Self-destructed Contracts: Finding whether a verified smart contract has been self-destructed is straightforward. If a contract has self-destructed, there will be a label (*Self Destruct*) given on Etherscan (see Figure 2). We found 756 self-destructed contracts from 54,739 verified smart contracts. The detailed information (creation time, destructed time, number of transactions / balances) of each contract can be found at https://zenodo.org/record/5518527#.YUl-bWYzYUE/Contract_Info_Suicide.csv. Table 1 shows a brief summary of these 756 contracts. Life span is the time interval between the

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:13

Fig. 2. A smart contract on Etherscan

Fig. 3. Transactions of a Self-destructed Contract

transaction of creation and destruction. No. Trans / Eths records the transactions and balance of a contract before its destruction, respectively.

Transactions: Transactions on Ethereum record the information of the external world interacting with the Ethereum network. All the transactions can be found on Etherscan. We collect

all 413,796 transactions of 756 self-destructed smart contracts. Figure 3 is the transactions of a self-destructed contract. In the first transaction, we can find who deployed the contract (creator) and we can find who destructed the contract (destructor) in the last transaction.

4.2.2 Stage 2. PS Pairs Generation: A *PS* (predecessor and successor) pair is denoted as: $PS = \langle \mathbb{P}, \mathbb{S} \rangle$, where $\mathbb{P}$ is a smart contract that has executed the selfdestruct function, named as the predecessor contract. $\mathbb{S}$ represents $\mathbb{P}$'s upgradeable version, that we call a successor contract. $\mathbb{P}$ and $\mathbb{S}$ are deployed by the same address and have similar functionalities.

The aim of stage 2 is to find all the PS pairs in our dataset, which helps to reduce the manual effort when summarizing the reasons for self-destruct in stage 3. For some self-destructed contracts, it is not easy to find their successor contracts. For example, in our survey feedback, some developers mentioned that they added the *selfdestruct* function is only to quickly transfer Ethers and will not redeploy the contract to Ethereum. For this kind of contracts, we cannot find their successor ones, and thus we removed them from our analysis list. Some developers said they would destruct the contracts when bugs are found. Then, they will redeploy a smart contract after fixing the bug. Thus, we can compare the difference of two versions of contracts to find the bugs, and the two versions of the contracts are likely to have high similarity. In this stage, we calculate the similarity to find the successor contracts of the self-destructed contracts. We found 436 contracts among 756 self-destructed contracts that have successor contracts.

Step 2.1 Cluster: We first find the creator addresses of all the 54,739 verified smart contracts through their transactions. In this step, we inspect the first transaction of each smart contract as the first transaction contains the creator address and creation time. Then, we classify the contracts into several groups according to their creator addresses. If two contracts have the same creator address, they will be classified into the same group. We only choose groups that contain self-destructed contracts.

Step 2.2 Rank by Time: A *PL* (predecessor and alive) pair is denoted as: $PL = \langle \mathbb{P}, \mathbb{L} \rangle$, where $\mathbb{P}$ is a smart contract that has executed the selfdestruct function. $\mathbb{L}$ represents a smart contract that has been deployed later than $\mathbb{P}$. $\mathbb{P}$ and $\mathbb{L}$ are deployed by the same address.

In this step, we first rank contracts in each group by their creation time, which can be obtained from the first transaction. Then, we can obtain several *PL* pairs. For example, one group contains five contracts, they are contract *a,b,c,d,e* and these five contracts are ranked by creation time. Contract b and d are the self-destructed in these five contracts. Finally, we output four PL pairs, i.e., *(b,c), (b,d), (b,e)* and *(d,e)*.

Step 2.3 Text Similarity: We compute the code similarity between two contracts to identify whether the later created contract is the successor contract of the self-destructed contract. SMARTEM-BED [42, 43] is the only tool that is specialized for calculating the similarity between smart contracts developed by Solidity at the time of writing this paper. According to their paper, SMARTEMBED obtains excellent performance in calculating the similarity of Ethereum smart contracts and outperforms the traditional similarity-checking / clone-detection tools, e.g., Deckard [47]. Thus, we use SMARTEMBED to calculate the code similarity instead of using other similarity checking techniques, e.g., vanilla [?].

SMARTEMBED first converts a smart contract into a code embedding by parsing the AST of a smart contract (details see Section 7) and then calculate the similarity between two contracts. The similarity metric is calculated as: $Similarity(c_1, c_2) = 1 - \frac{Euclidean(e_1, e_2)}{||e_1|| + ||e_2||}$, where c_1 and c_2 are two smart contracts; e_1 and e_2 are their corresponding code embeddings, which can be presented as $e_i = \{w_{i_1}, w_{i_2}, ..., w_{i_n}\}$. Euclidean function is used to point the distance between e_1 and e_2, which is calculated by $Euclidean(e_1, e_2) = \sqrt{(w_{1_1} - w_{2_1})^2 + (w_{1_2} - w_{2_2})^2 + ... + (w_{1_n} - w_{2_n})^2}$. Although the tool is aimed at finding bugs, their first step computes code similarity between the given smart

contract and history contracts. We modified the source code of SmartEmbed to compute the similarity between two contracts. If their similarity is larger than 0.6, they might be relevant and we assume the later created contract is the successor of the self-destructed contract. We also called this self-destructed contract as the predecessor contract of the successor contract. We found 436 self-destructed contracts have their successor contracts with 1513 *<predecessor contract, successor contract>* pairs. We note that 0.6 is a conservative threshold (original paper assumes similarity 0.95 are cloned); we might include many irrelevant pairs in our dataset, but it will not influence our result as we conduct a manual analysis in the subsequent step. Increasing the threshold can remove some irrelevant pairs to reduce the manual effort, but it might make us miss some true matching pairs. Besides, some token contracts might have duplicated code with high similarity, but it still will not affect the results. Because we will analyze the difference of the similar token contracts manually to identify whether the later created contracts are the successor contracts of the prior contracts.

4.2.3 Stage 3. Reason Generation: In stage 2, we found 436 contracts among 756 self-destructed contracts that have successor contracts. Note that the PS pairs we found in stage 2 might contain many false positives. For example, two contracts can obtain very high code similarity if they use many common open-source libraries, but they are not the valid PS pairs. Thus, we need manual analysis to remove these false positives.

It is a time-consuming and error-prone process to analyze the predecessor contract and successor contract directly. To deal with these issues, we perform two steps. First, to reduce the manual effort, we use a tool named *DiffChecker* [7] to help us find the difference between the predecessor and successor contract. The basic idea of *DiffChecker* is to use the Longest Common Substring (LCS) [39] algorithm to find the longest string (or strings) that is a substring (or are substrings) of two or more strings. We use *DiffChecker* to highlight the difference to reduce manual efforts. Second, to increase the reliability of our results, we conduct open card sorting to summarize the reason why a smart contract self-destructed. Guided by previous works [26, 61], we create one card for each *PS* pair. Each card highlights the difference between the two contracts.

The detailed steps of the open card sorting we used are:

Iteration 1: We randomly chose 20% of the cards, and two developers with 3 years of smart contract development analyzed the difference of the code and discussed the reason why contracts self-destructed. They first quickly read the two contracts to identify whether they are relevant (The two contracts have similar functionalities). If they are irrelevant, the card will be discarded. Then, they carefully read the difference between the contracts and discuss the reason for this difference. For example, Figure 4 is a real example of a predecessor contract (left) and its successor contract (right) in our dataset. The three differences between the two contracts are highlighted. First, the developer added a *Transfer* event in Line 356 of the successor contract. Second, in the predecessor contract, Ethers can only be sent to an address whose balance is zero. This restriction was removed in the successor contract. Finally, the *selfdestruct* function was also removed. According to our definition, all of these three modifications change the code representation of the contract. Thus, the reason for the Self-destruct is regarded as *Functionality Changes*. For some contracts, it is not easy to find the reason for the self-destruct usage and they were omitted from our card list. All the reasons are generated during the sorting.

Iteration 2: The same two smart contract developers independently categorized the remaining 80% of the cards into the initial classification scheme. Then, they compared their results and discussed disagreements. We used Cohen's Kappa [32] to measure the agreement between the two developers. Their overall Kappa value is 0.84, indicating strong agreement.

```
353    function ZipToken() public {                   353    function ZipToken() public {
354        totalSupply_ = INITIAL_SUPPLY;             354        totalSupply_ = INITIAL_SUPPLY;
355        balances[msg.sender] = INITIAL_SUP         355        balances[msg.sender] = INITIAL_SUP
    PLY;                                              PLY;
                                                      356        Transfer(0x0, msg.sender, INITIAL_
                                                          SUPPLY);
356    }                                              357    }
357                                                   358
358    function distributeTokens(address[] ad         359    function distributeTokens(address[] ad
    dresses, uint[] values) public onlyOwner {           dresses, uint[] values) public onlyOwner {
359        require(addresses.length == value          360        require(addresses.length == value
    s.length);                                           s.length);
360        for (uint i = 0; i < addresses.len         361        for (uint i = 0; i < addresses.len
    gth; i++) {                                           gth; i++) {
361            address a = addresses[i];              362            address a = addresses[i];
362            uint v = values[i];                    363            uint v = values[i];
363            if (balanceOf(a) == 0) {               364            transfer(a, v);
364                transfer(a, v);
365            }
366        }                                          365        }
367    }                                              
368                                                   
369    function die() public onlyOwner {              
370        selfdestruct(msg.sender);                  
371    }                                              366    }
372 }                                                 367 }
```

Fig. 4. An example considered in our card sort that contain a predecessor contract (0x3b96990a8ef293cdd37c8e1ad3d210a0166f40e1) and successor contract (0xedd7c94fd7b4971b916d15067bc454b9e1bad980).

Table 2. Reasons of Self-destruct and their distributions among 340 self-destructed smart contracts.

Category	Description	Distribution
Functionality Changes	Adding, removing or changing Functionalities for upgrading contracts to respond new requirements. Functionality changes will change the code representation of a contract, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG).	156 (45.88%)
Limits of Permission	Adding permission checks for the sensitive functions.	25 (7.35%)
Unsafe Contracts	Removing the security problems of the contracts.	95 (27.94%)
Unmatched ERC20 Token	Modifying the contract to make it follows the ERC 20 standard	19 (5.59%)
Setting Changes	Changing the variable or function states of the contracts, such as renaming a contract, changing the amount of ERC20 token supplement, changing a public function to private function. Setting Changes will not remove or add new code from a contract.	56 (16.47%)

We finally identified 5 reasons why contracts have been self-destructed. This information is shown in Table 1 and the detailed information is shown in the following subsection.

4.3 Reasons for Self-destruct

In this subsection, we give detail explanations of the 5 self-destruct reasons and their distribution in our dataset.

4.3.1 Definitions. The short descriptions of 5 self-destruct reasons are given in the first two columns of Table 2. Below we give a detailed description each reason.
(1) Functionality Changes: Due to the immutability of smart contracts, it is not easy to upgrade smart contracts. However, during the entire smart contract life cycle, it is necessary for developers to add, remove or change some functionalities to respond to the new requirements. Functionality changes will change the code representation of a contract, e.g., Abstract Syntax Tree (AST), Control

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:17

Fig. 5. A real diff example of Limits of permission in our dataset. Predecessor Address: 0xfa1d63b87f40c92d27bfb255419c1ea8c49086de; Successor Address: 0x64b09d1a4b01db659fc36b72de0361f2c6c521b1

Fig. 6. A real diff example of Unsafe Contract in our dataset. Predecessor Address: 0x8b099bdcfea93faecfac13d0dbc1d08c4e1ec595; Successor Address: 0x17683235257f2089e3e4acc9497f25386a529507

Fig. 7. A real diff example of Unmatched ERC20 token in our dataset. Predecessor Address: 0x848217a9569ca64fffba9d000cda05f9d2fa97f5; Successor Address: 0xf42230a7e21375c29648ae9544f7da394e20ead3

Flow Graph (CFG). According to our analysis, we find that functionality changes are the most common reason why smart contracts are self-destructed. When new requirements appear or some requirements are changed, some developers choose to deploy a new smart contract and the old version of the contract will be destructed.

(2) Limits of Permission: Ethereum is a permission-less network [27] and anyone can call the functions of the contracts by sending a transaction. Thus, it is important to limit the access permissions for some sensitive functions. According to our analysis, we find some predecessor contracts do not check permissions of the callers in some sensitive functions, e.g., Ether transfer. Thus, everyone can execute the sensitive functions. In the successor contract, they add permission checks to limit access permissions.

   Example: Figure 5 is a real example of Limits of Permission. The predecessor contract does not limit the permission of calling the selfdestruct function. In the successor contract, the function adds a modifier *onlyOwner* to check the permission. Thus, only the contract owner can call this function.

(3) Unsafe Contracts: Previous works [26, 48, 52, 56, 62] highlighted several security problems of smart contracts. For example, *Oyente* highlighted four security issues, *Zeus* described four security problems of smart contracts (see Section 7). We find many predecessor contracts contain security problems like *reentrancy*, which can lead to Ether loss. Developers usually fix these security issues in the successor contracts.

   Example: Figure 6 is a real example of Unsafe Contracts. The predecessor contract does not check the return value of the *msg.sender.send()*, which might lead to security issues of the contract. In the successor contract, the function checks whether the Ether send is successful. If not, the transaction will be thrown.

(4) Unmatched ERC20 token: ERC20 [3] is the most popular standard interface for tokens in Ethereum. If the implementation of token contracts does not follow the ERC20 standard strictly, the transfer between tokens may lead to errors. We find many predecessor contracts are token contracts but do not strictly follow the ERC20 standard, while their successor contracts do follow the standard.

Fig. 8. A real diff example of Setting Changes in our dataset. Predecessor Address: 0xa41aa09607ca80ee60d2ce166d4c02a71860e5c5 ; Successor Address: 0x41c6af7b388e80030e63f2686dc2ff9bfd1267c9

Example: ERC20 requires a transfer function to return a boolean value to identify whether the transfer is successful. However, the transfer function in the predecessor contract in Figure 7 does not return anything. Users usually use third-party tools to manipulate their tokens and these tools capture token transfer behaviors by monitoring standard ERC20 method [30]. If the contract does not match the ERC20 standard, the token may fail to be transferred by third-party tools. In the successor contract, the return value of the *transfer()* function is added.

(5) Setting Changes: Similar to *Functionality Changes*, it is likely that developers will change some settings of smart contracts in response to new requirements. For example, the token-related contracts usually have some default values, e.g., total token supply, number of decimals, token name. Due to immutability, if developers want to change the total token supply, they have to destruct the contract and deploy a new one. The main difference between *Setting Changes* and *Functionality Changes* is that *Setting Changes* will not add or remove code from contracts. Thus, the structure of AST and CFG should be the same between two contracts, if the reason for their selfdestruct is *Setting Changes*.

Example: Figure 8 is a real example of Setting Changes. The token supplement in the predecessor contract is too small. Thus, the successor contract changes the token supplement by multiple a value DECIMALS (DECIMALS equals to 10^{18} in the successor contract.)

4.3.2 Distribution. We use SMARTEMBED to find the pair*<predecessor contract, successor contract>*, and set the threshold value to 0.6, which might obtain some irrelevant pairs. After manually removing irrelevant pairs, we found 340 contracts (**some contracts have multiple self-destruct reasons. If multiple changes are found in the successor contract, we regard all of the changes as contributing to the self-destruction.**) for which we can identify the reason(s) why they are self-destructed and give the distribution of the five self-destruct reasons in the last column of table 2. 96 contracts cannot find the reason why they are self-destructed, and thus they are omitted from our dataset.

It is clear that *Functionality Changes, Unsafe Contract*, and *Setting Changes* are the top three most popular reasons that lead to contracts destructed; the number are 156, 95 and 56, respectively. The number of the other two reasons are similar, there are 25 contracts destructed for *Limits of Permission* and 19 for *Unmatched ERC20 Token*.

It should be noted that it is not easy to find all the security issues in our dataset. One the one hand, we only checked the security issues reported by *Oyente, Zeus, Mythril, Securify, Maian* [5, 48, 52, 56, 62]. On the other hand, manually checking for security issues is very error-prone and time-consuming. To reduce the errors, we utilized tools by *Oyente, Zeus, Mythril, Securify, Maian* and manually checked each contract it found. We first use the tools to check smart contracts. Then, two developers with 3 years of smart contract development experience manually identified whether the results are correct. If the reported results were different, they discussed to obtain the final result.

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:19

Note that the code of *Oyente, Mythril, Securify, Maian* can be found on Github, and we rerun the tool to get the result. We did not find the code of *Zeus*, but *Zeus* provides their evaluation results which inform whether a contract address contains vulnerabilities or not. Thus, we use their evaluation results directly. If the detected contract does not appear in their evaluation results, we regard Zeus as not finding any vulnerabilities in the contract.

5 RQ3: LIFESCOPE: A SELF-DESTRUCT ISSUES DETECTION TOOL FOR SMART CONTRACTS

5.1 Motivation

In the previous section, we introduced five smart contract self-destructed reasons by comparing the difference between predecessor contracts and their successor contracts. Among these five reasons, *Functionality Changes* and *Change Setting* depend subjectively on the contract owner's requirements. Specifically, different developers might make different decisions of whether a smart contract should be self-destructed according to their requirements, even if the smart contracts are the same. It is thus hard to say these two reasons can affect the life cycle of smart contracts. However, smart contracts that contain the other three self-destruct reasons might have a short life span, as they can lead to unwanted behaviors of the smart contracts. Detecting whether a smart contract contains these self-destruct reasons might increase the life span of the contract. Manual analysis is time-consuming and error-prone. Therefore, designing a tool to detect whether a contract contains these self-destructed reasons before deploying them to the Ethereum is important. Security issues is a big concept. In the last section, we use the security vulnerabilities defined in previous works, e.g., *Oyente, Zeus, Mythril, Security, Maian* [5, 48, 52, 56, 62], to find unsafe contracts. These have already proposed several tools to detect security issues with high accuracy. The accuracy of *Zeus* is almost 100% according to their paper, and designing a more accurate and comprehensive security detecting tool is not the main target of this paper. In this case, we do not redevelop a tool to detect security issues introduced in these previous works.

5.2 Approach

We propose a tool named *LifeScope* to detect the remaining two issues, i.e., *Limits of Permission* and *Unmatched ERC20 Standard* that can lead to contracts being destructed. Since the aim of *LifeScope* is extending the life span of a smart contract by finding the self-destruct reasons, and smart contracts are immutable to be modified after deploying to the blockchain. Therefore, it is meaningless to detect the two self-destruct reasons through bytecode, although smart contracts are stored in the form of bytecode.

```solidity
function totalSupply() public returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (
    bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256
    remaining)
event Transfer(address _from, address _to, uint256 _value)
event Approval(address _owner, address _spender, uint256 _value)
```

Listing 3. ERC20 Functions and Events

LifeScope detects the self-destruct issues at source code level, which utilizes AST (abstract syntax tree) to parse the smart contracts and extract related information to detect *Unmatched ERC20 Standard*. For *Limits of Permission*, *LifeScope* first transfers the contract to a TF-IDF representation

---

Algorithm 1: Algorithm to Detect Unmatched ERC20 Token

  Input: AST of a smart contract
  Output: Is an Unmatched ERC20 Token

1 Extract $\langle fun_n, funInfo \rangle$ pair list from AST;

2 Extract $\langle event_n, eventInfo \rangle$ pair list from AST;

3 appearedFunc = 0;

4 legalFunc = 0;

5 legalEvent = 0;

6 isMachedERC20 = false;

7 **for** $fun_i, funInfo_i \in \langle fun, funInfo \rangle$ *pair list* **do**

8   **if** fun_i *is one of ERC20 Standard Function* **then**

9   appearedFunc++;

10   **if** $funInfo_i$ *is same to ERC20 Standard Function* **then**

11   legalFunc++;

12   **end**

13   **end**

14 **end**

15 **for** $event_i, eventInfo_i \in \langle event, eventInfo \rangle$ *pair list* **do**

16   **if** $event_i$ *is one of ERC20 Standard Event* **then**

17   **if** $eventInfo_i$ *is same to ERC20 Standard Event* **then**

18   legalEvent++;

19   **end**

20   **end**

21 **end**

22 **if** *appearedFunc < 5* **then**

23   return not_ERC20_Contract;

24 **end**

25 **if** *legalFunc == 6 and legalEvent == 2* **then**

26   isMachedERC20 = true;

27 **end**

28 return isMachedERC20;

---

and then utilizes machine learning algorithms to predict this problem. These two problems are not only limited to contracts that contain the *selfdestruct* function. Any smart contracts can be analyzed with *LifeScope* to detect these two problems before deploying them to the Ethereum.

5.2.1 Unmatched ERC20 token. The method to detect the *Unmatched ERC20 Token* is shown in Algorithm 1. The input is an AST (Abstract Syntax Tree) of a smart contract, which is generated by the *Solidity* compiler [17]. AST is a tree structure that contains the syntactic information of source code. By analyzing the AST, we can generate a list of pairs $\langle fun_n, funInfo \rangle$. fun_n is the name of a function; $funInfo$ contains information about the function fun_n, i.e., parameter types and return types. ERC20 standard defines nine functions and two events. Among the nine functions, three are optional, and six are compulsory. The six compulsory functions and two events are shown in Listing 3. We traverse all the functions in the smart contract. If the function name fun_n is one of the six compulsory functions, we then compare whether the input parameter types and return types are

Why Do Smart Contracts Self-Destruct?
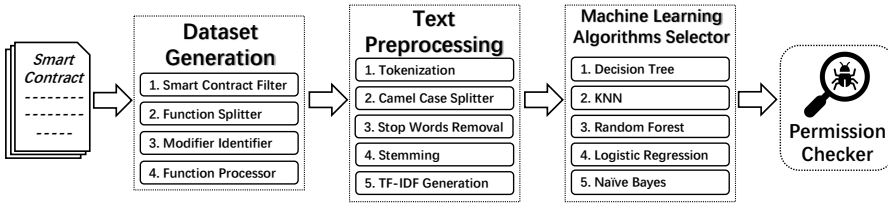Investigating the Selfdestruct Function on Ethereum

1:21

Fig. 9. Approach to Check Permissions

the same with the ERC20 standard. For example, if a contract contains a function named *transfer*, we then check whether this function contains two parameters and their types are *address* and *uint256*, respectively. Besides, the function should have a return value, and the type of the return value is *bool*. We use the same method to check contract events. Finally, if all the six compulsory functions and two events appear in the contract, this is a matched ERC20 smart contract. We follow the previous work [41], if less than five ERC20 functions appear in a smart contract, we regard it as not an ERC20 smart contract. Otherwise, it is an unmatched ERC20 token. Note that Solidity allows abstract functions or interfaces which do not have code implementation. We only consider a function as an ERC20 function if it has complete implementation. Besides, the bytecode of a public storage variable is the same as a view function without any input parameters and only has one return value. Thus, the function *totalSupply()* can also be represented as a global variable, i.e., *uint totalSupply = amount;* If a smart contract does not contain a function named *totalSupply()*, but contains a public variable named *totalSupply*, it will also be regarded as a matched ERC20 function.

5.2.2 Limits of Permission. It is hard to prescribe functions need to check their permissions. Therefore, It is not easy to detect this issue by using programming analysis methods. We utilize a machine learning method to predict whether a function needs to check for its caller permission. Figure 9 describes the overall architecture that we used. The method contains three parts, i.e., *Dataset Generation*, *Text Preprocessing* and *Machine Learning Algorithm Selector*.

(a). *Dataset Generation:* The aim of this step is to extract pairs*<func, permission>* from smart contracts. In each pair, *func* is the source code of a function in the smart contract, and *permission* means whether the function needs to check the caller's permission. Since security vulnerabilities are ubiquitous in smart contracts on Ethereum [31, 48, 52, 56], some alive contracts might also miss checking the permissions of the functions. This situation gets even worse in self-destructed contracts, as the reason for the destruct might be missing permissions. Therefore, it is not reliable to use these contracts as our ground truth. To ensure the correctness of our dataset, we should use contracts that correctly check their permissions for the contracts. However, it is not easy to ensure the correctness of the contract, and manually check whether a function needs to check its permission is also error-prone and subjective.

To obtain the dataset, we first rank all the alive verified contracts by their transaction numbers. We then choose all of the contracts whose transaction numbers are larger than 500, as transactions can be regarded as test cases for the contract. Previous works check the transaction input to detect malicious attacks [28, 38, 66]. The more the number of normal running transactions a contract has, the less likely the contract has permission problems. After this step, 5,986 contracts remained. Financial gain is an important motivation behind the attacks on Ethereum smart contracts [27]. We finally find 1 Ether can get an appropriate number of the dataset (875 smart contracts with 29,313 functions). Thus, we choose contracts whose balance have more than 1 Ether (1 Ether worths about $1400 at Feb. 2021). The sensitive information of smart contracts, e.g., contract bytecode,

balance, are visible to the public. Finally, 875 contracts whose balance has larger than 1 Ether and transaction numbers larger than 500 transactions remain.

After getting these contracts, we first remove the comments on the contracts and then split the contracts into functions. We obtain 29,313 functions from these 875 smart contracts. We then need to identify whether these functions contain modifiers, as the permission is usually checked by the modifier. If a function contains modifier, we remove the modifier from the function. For example, the original function is *function transferMoney(address addr) onlyOwner{}*. We remove the modifier *onlyOwner* from the function and only use *function transferMoney(address addr){}* to training the machine learning model. In our dataset, we obtain 29,313 functions, with 4,393 of them needing to check permissions.

(b). Text Preprocessing: Before training the machine learning module, we need to transfer each processed function into a set of bag-of-words (BoW). The dataset is split into a training set and a test set. Both of them are first processed by the following steps: **(i) Tokenization**: Each processed function is divided into a list of words by punctuation and space that usually do not contain any information. For example, *function transferMoney(address addr){}* will be transferred into "function", "transferMoney", "address" and "addr" **(ii) Camel Case Splitter:** We separate function names, variable names and identifiers according the rules of Camel Case [20]. For example, "transferMoney" will be separated into "transfer" and "Money". Then, we transfer all the words to their lower case. **(iii) Stop Words Removal:** Stop words means meaningless words (e.g., "to", "as", "is"). We adopt NLTK (a python library) stop words list in this step. We also remove tokens of less than 3 characters. **(iv) Stemming:** this step is used to transfer words into their stem form. For example, "running" is replaced by "run". In this paper, we use Porter's stemmer [59] to transfer the words.

After these four steps, we use TF-IDF (term frequency - inverse document frequency) [21] to represent each processed word in the function. (Noticed that to make the result more reliable, the BoW model is only built by the training set.) This is described as:

$$w_{i,j} = tf_{i,j} * log(\frac{\#of functions}{df_i}) \tag{1}$$

Here, $w_{i,j}$ is the weight of the word i in the function j. $tf_{i,j}$ is the term frequency of word i in the function j. df_i is the number of functions that contain word i. Finally, each function is represented as $fun_j = (w_{1,j}, ...w_{i,j}, ..., w_{n,j})$.

(c). Machine Learning Algorithms Selection: Checking the permission of functions is a binary classification problem. We tried five popular machine learning algorithms to find an appropriate algorithm for predicting the permission and use the algorithm that obtains the best F-Score for this task. The five algorithms are *Decision Tree, KNN, Random Forest, Logistic Regression*, and *Naive Bayes*. We use a python library named sklearn [?] with its default configuration to implement these five algorithms.

5.3 Evaluation For Unmatched ERC20 Token

5.3.1 Dataset. The dataset for *Unmatched ERC20 token* consists of two parts, i.e., smart contracts with and without *selfdestruct* function. The dataset with *selfdestruct* function has 756 self-destructed contracts, which is the same dataset introduced in RQ2. Among these contracts, 127 of them are discarded due to the unsupported compiler version. (LifeScope supports Solidity compiler versions that are equal to or higher than 0.4.25). So, there are finally 629 self-destructed contracts in our dataset. The two researchers manually labeled the dataset and found 164 of them are ERC 20 token contracts. In these 164 ERC20 token contracts, 70 (11.13%) of them are *Unmatched ERC20 tokens*.

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:23

For the dataset without *selfdestruct* function, we use an open source dataset proposed by our previous work [26]. The dataset contains 587 smart contracts, which has the ground truth of *Unmatched ERC20 tokens*. The ground truth is labeled by two experienced researchers. They first analyzed the contracts independently, and then discussed any differences, which ensured the correctness of the dataset. Since the dataset is randomly selected, some contracts contain the *selfdestruct* function and a low compiler version. After removing the incompatible contracts, 358 contracts remained. Among these 358 smart contracts, 141 are ERC20 token contracts, and 36 (25.53%) of them have an Unmatched ERC20 token.

5.3.2 Result. For the dataset with *selfdestruct* function, *LifeScope* finds 70 *Unmatched ERC20 token*, with 0 false positive and negative. For the dataset without *selfdestruct* function, LifeScope finds all 36 Unmatched ERC20 token, with 0 false positives and negatives. The results show that LifeScope can also detect the *Unmatched ERC20 token* problem in both smart contracts with and without *selfdestruct* functions.

5.3.3 Comparison: TokenScope [30] is a novel transaction based tool to identify the inconsistency of ERC20 tokens. It can identify whether a contract is a legal ERC20 token by investigating its transactions. In this paper, we re-execute *TokenScope* and use the same dataset to compare with LifeScope.

The dataset with a *selfdestruct* function contains 164 ERC20 contracts. Among these contracts, 94 are legal ERC20 contracts, and 70 have an unmatched ERC20 Token. *TokenScope* correctly predicts 88 contracts are legal ERC20 tokens, and 49 are unmatched ERC20 tokens. However, it mistakenly predicts 6 contracts are not legal ERC20 tokens, and 21 are ERC20 tokens.

The dataset without *selfdestruct* function contains 141 ERC20 contracts. Among these contracts, 105 are legal ERC20 contracts, and 36 are unmatched ERC20 Tokens. *TokenScope* correctly predicts 96 contracts are legal ERC20 tokens, and 17 are unmatched ERC20 tokens. However, it mistakenly predicts 9 contracts are not legal ERC20 tokens, and 19 are ERC20 tokens.

TokenScope needs transactions to identify whether a contract has legal ERC20 tokens. However, since some contracts only have a signal transaction, this leads to the false positives of *TokenScope* (mistakenly predicting a contract is not an ERC20 token). Besides, we find *TokenScope* cannot check the return value of a function. Specifically, ERC20 standard requires the transfer function to return a boolean value, while *TokenScope* cannot identify whether a function has a return value, which leads to false negatives (mistakenly predicting a contract is an ERC20 token).

In conclusion, *LifeScope* performs better than *TokenScope*.

Etherscan also can identify whether a contract is an ERC20 contract. We did not compare the result with Etherscan because we have different standards to define whether a contract is an ERC20 token. ERC20 standard defines nine functions and two events. Among the nine functions, three are optional, and six are compulsory. For our paper, we use a definition in previous work [41]: if less than five compulsory ERC20 functions appear in a smart contract, we regard it as not an ERC20 smart contract. A smart contract is defined as an unmatched ERC20 contracts only if it has more than or equal to five compulsory ERC20 functions, and some of these functions do not follow ERC20 standards. However, Etherscan has a different definition for ERC20 contracts. Etherscan regards a contract that has ERC20 related transactions as an ERC20 contract even though it only contains one ERC20 function, e.g., *transfer()* and has related transactions. Thus, the result of Etherscan will be very different from our method, and it is the reason why we did not choose it as our comparison method.

Table 3. Results of Predicting *Limits of Permission* by using Five Machine Learning Algorithms

	Precision	Recall	F-Measure	Accuracy	AUC
Decision Tree	78.91%	77.09%	77.89%	93.45%	0.8673
KNN	72.75%	50.40%	59.50%	89.71&	0.7352
Random Forest	83.73%	70.82%	76.05%	94.88%	0.8499
Logistic Regression	84.11%	52.78%	64.82%	91.40%	0.7551
Naive Bayes	23.33%	88.43%	35.66%	52.15%	0.6708

Table 4. Results of Predicting *Limits of Permission* for 10-fold cross-validation by using Decision Tree

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG
Precision	79.92%	79.08%	77.45%	76.76%	77.14%	77.04%	79.20%	77.32%	82.96%	82.19%	78.91%
Recall	69.08%	73.14%	77.28%	88.19%	82.50%	78.40%	81.34%	76.93%	76.84%	73.39%	77.09%
F-Measure	74.10%	76.00%	77.37%	79.29%	79.73%	77.72%	80.26%	77.13%	79.78%	77.54%	77.89%
Accuracy	92.35%	93.16%	93.72%	93.97%	94.01%	93.45%	93.88%	93.33%	93.72%	92.95%	93.45%
AUC	0.8291	0.8489	0.8682	0.8896	0.8921	0.8721	0.8874	0.8653	0.8690	0.8511	0.8673

5.4 Evaluation For Limits of Permission

5.4.1 Dataset. The method that was used to generate the dataset of *Limits of Permission* is introduced in section 5.2. To better evaluate the results, we use a cross-validation method of training-testing sets. First, we divided our dataset into 10 parts of equal sizes. Then, we conduct the training using 7 parts of the dataset and 3 parts for testing. Specifically, we give an ID (0 to 9) to each part. In the first round, the parts with ID 0 to 2 are the testing sets, and 3-9 are the training sets. In the last round, the parts with ID 9, 0, 1 are the testing parts, and the remaining parts are the training set. We continue this process 10 times. Finally, we report the average results. There are around 20,328 case in our training sets and 8985

5.4.2 Evaluation Methods and Metrics. We use five measurements to evaluate the results, i.e., precision, recall, F1-Measure, accuracy, and AUC. Precision, Recall, F-measure, and Accuracy can be calculated as: $\frac{\#TP}{\#TP+\#FP}$, $\frac{\#TP}{\#TP+\#FN}$, $\frac{2\times P\times R}{P+R}$, $\frac{\#TP+\#TN}{\#TP+\#TN+\#FN+\#FP}$, respectively. TP (true positive) indicates the number which correctly predicts a function needs to add a permission check. TN (true negative) indicates the number which correctly predicts a function does not need to add a permission check. FP (false positive) and FN (false negative) indicate the number which incorrectly predicts that a function needs or does not need to add a permission check. AUC (area under the curve) is calculated by plotting the ROC curve (receiver operator characteristic).

5.4.3 Result. Table 3 shows the result of predicting *Limits of Permission* by using five machine learning algorithms. We found that the *Decision Tree* algorithm obtains the best F-Score for this task. Therefore, we finally use the *Decision Tree* to predict whether a function needs to check for its callers' permission.

The detailed results of predicting *Limits of Permission* for 10-fold cross-validation by using Decision Tree are shown in Table 4. Our method obtains 78.91% of precision, 77.09% of recall, 77.89% of F-measure, 93.45% of Accuracy, and 0.8673 of AUC. Prior works [36, 37] suggest that a classifier performs reasonably well if its AUC is larger than 0.7.

5.4.4 Comparison. To evaluate the performance of *LifeScope* in detecting *Limits of Permission*, we design a keywords-based method to identify whether a function needs to check its permission. Following we describe the details of the comparison method.

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:25

Feature Selector: We use decision tree to predict whether a function needs to add a permission check, which enable us to know which words lead to a function being classified as needing permission check. Guided by previous works [45, 60, 69], we employ a widely used feature selection technique named Information Gain, to select useful features in our prediction.

Our dataset can be denoted as $F = (F_1, L_1), (F_2, L_2), ..., (F_n, L_n)$, where F_i represents the ith function, and L^i is the label, which means whether the F_i needs to check its permission (t) or not ($\bar{t}$). The word vector of F_i is represented as $F_i = \{w_1, w_2, ..., w_n\}$, where n represents the number of different words appeared in F_i, and w_i represents the ith words. There are four relationships between the word w and a function F_i.

1. (w, t): function F_i contains the word w, and the function needs to check its permission.
2. $(w, \bar{t})$: function F_i contains the word w, but the function does not need to check its permission.
3. $(\overline{w}, t)$: function F_i does not contain the word w, but the function needs to check its permission.
4. $(\overline{w}, \bar{t})$: function F_i does not contain the word w, and the function does not need to check its permission.

Based on these relationships, the information gain (IG) of word w' and label t' is defined as:

$$IG(w, t) = \sum_{t' \in \{t, \bar{t}\}} \sum_{w' \in \{w, \overline{w}\}} p(w', t') \times log \frac{p(w', t')}{p(w') \times p(t')} \qquad (2)$$

In the equation, $p(w', t')$ is the probability of the word w' in a function with label t'. $p(w')$ means the probability of word w' in a function and $p(t')$ represents the probability of a function with label t'.

Behaviors of permission check: Information gain reflects the amount of information required for predicting a label (needs or does not need to check the permission). The higher IG score a word has, the more important the word to distinguish the label. We rank the IG score of each word and list the top 50 words with highest information gain score in Table 5. The first line has the highest IG score, and the score decreases from left to right in each line. Note that a word with a high IG score means it has a high contribution to predicting the label, not necessarily mean that it indicates whether a function needs permission check. Thus, we manually check the word in Table 5 to find useful behaviors.

From the words, we then summarize four kinds of behaviors that needs to check functions' permission:

(1). **Ether Transfer.** Specifically, the words "msg", "sender", "transfer", and "eth" are related to Ether transfer methods on Ethereum, i.e., *msg.sender.transfer(eth)*; the word "withdraw" reflects a behavior related to withdraw balance.

(2). **Sensitive states change**, e.g., changing permission owners, increasing or decreasing total supply of tokens, stopping or starting the functionalities. Specifically, the words "ownership", "administr", "mint" and "renounce" are all related to sensitive states of a contract.

(3). **Inline assemble and selfdestruct function.** Ethereum provides some functionalities which need to limit permission. Specifically, the words "mload" and "assemble" are related to the inline assemble. The word "selfdestruct" is related to the *selfdestruct* function.

(4). **Emergency Management.** Smart contracts are difficult to be modified once deployed. Thus, there are some functions used to handle the emergency situations. Specifically, the words "emerge", "pause", "stop", "unpause", "selfdestruct" are related to emergency management.

We use the top 50 words with the highest IG score to find four behaviors that need to check functions' permission. To prove the completeness of the four behaviors, we also manually check the top 51-100 words with the highest IG score. We find there are no additional behaviors that can be found. All the words can be classified into these four categories.

Table 5. Top 50 words with highest information gain score.

ownership	transfer	name	oracle	pidx	gen	address	pause	human	round
stop	unpause	assert	online	data	addr	core	log	compress	assemble
withdraw	gap	event	mint	mask	eth	within	msg	transaction	mload
earn	emerge	spender	code	sender	renounce	determine	calc	sqrt	team
propose	math	divest	div	selfdestruct	filter	administr	confirm	investor	sale

Result of the Comparison Method: In the last part, we summarize four kinds of behaviors that need to check functions' permission according to the top 50 words with the highest IG sore. According to these words, we design a simple keywords based method to identify whether a function needs to check its permission. Specifically, if a function contains one of the key word "msg.sender.transfer", "ownership", "administr", "mint", "renounce", "mload", "assemble" "emerge", "pause", "stop" and "unpause", we assume the function needs to check its permission. Notice that we merge some keywords, e.g, "msg", "sender", "transfer" to "msg.sender.transfer" as "msg.sender" are widely used in Ethereum. We finally obtain 31.53% precision, 11.41% recall, 16.75% F1-measure, and 83.01% accuracy, which are much worse as compared to our machine learning based method (78.91% precision, 77.09% recall, 77.89% F1-Measure, and 93.45% accuracy).

6 DISCUSSION

We first summarize the key implications of our work for researchers, practitioners, and educators. Then, we give 6 suggestions on how to better use *selfdestruct* function according to the feedback of the survey. Finally, we summarize the main threats of validity.

6.1 Implications

6.1.1 For Researchers. **Research Guidance.** In this paper, we found 5 reasons why smart contracts destructed by comparing the difference between self-destructed contracts and their successor contracts. With the increasing number of smart contracts, researchers can apply our methods to find more problems that can affect the life span of smart contracts. Our study focuses on Ethereum smart contracts, but many other blockchain platforms also support the running of smart contracts, e.g., Ethereum Classic [10], Expanse [11]. Both Ethereum Classic and Expanse were created by the hard fork of Ethereum, but currently they are independent blockchain systems with many years of development. Both of them support the running of smart contracts based on EVM and support the *selfdestruct* function. There might be some different reasons for the self-destruct; researchers can use our methods to identify the reasons on these platforms.

6.1.2 For Practitioners. Our work is the first that uses an online survey to collect feedback from smart contract developers on why they include or exclude *selfdestruct* function. Their feedback shows that adding a *selfdestruct* function can help developers transfer Ethers when emergency situations happen. However, using this function can also lead to several problems. To address the drawbacks of adding a *selfdestruct* function, we give 6 suggestions in the next section. These can help developers better use the *selfdestruct* function in their contracts. Smart contract developers can develop a smart contract according to our suggestions and open source the code for other developers to use. We also summarized 5 common reasons why contracts self-destructed and developed the LᴉꜰᴇSᴄᴏᴘᴇ tool to detect 2 self-destruct reasons. Removing these problems might extend the life span of smart contracts.

6.1.3 For Educators. *selfdestruct* is an important feature of smart contracts. However, most blockchain tutorials focus on teaching how to develop smart contracts and knowledge about blockchain [26].

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:27

Educators should pay more attention to these unique functions of smart contacts. For example, educators should mention the importance and drawbacks of adding a *selfdestruct* function when they introduce this function. The feedbacks of our survey in RQ1 can provide good materials for them.

6.2 Towards More Secure selfdestruct Functions

In Section 3, we summarized six reasons why smart contract developers exclude the *selfdestruct* function from their contracts. In this part, we give six suggestions about how to better use *Selfdesturct* function according to the summarized worries.

Suggestion 1. Limit Usage Scenario: Adding *selfdestruct* function can increase the complexity of the development and risk of attacks. Some smart contract developers claim that the *selfdestruct* function is mainly used to remove the code and transfer Ethers. However, they do not need this function if there is no Ether in their contracts. Removing the contracts from the blockchain is also unattractive to them. Even if their contracts are attacked and controlled by attackers, they can discard the old contracts and deploy a new version. To reduce the risk and the workload of development, a *selfdestruct* function is better to add in contracts that contain Ethers.

Suggestion 2. Permission Check: Calling a *selfdestruct* function can lead to irreversible consequences. The contract has to check the permission of the caller in each transaction. A common method to check the permission is recording the owner's address in the constructor function. Then, checking whether the caller is the owner in each transaction.

Suggestion 3. Distribute the Rights and Modularization: The trust concern is an important reason why developers exclude a *selfdestruct* function. The trust concern contains two parts according to the feedback, i.e., human related and code related concern.

For human related concern, users might worry that the owner of the contract can destruct the contract and transfer all the balance if the contract has a *selfdestruct* function. For example, the *gambling* contract shown in Listing 1 claims that users can transfer 1 Ether to the contract. When the contract receives 10 Ethers, the contract will choose one user as the winner and transfer 9 Ethers to the winner as a bonus. However, if the contract has a *selfdestruct* function, users might worry that the owner might transfer the money out at any time.

To reduce this kind of concern, the owner could build a DAO (Decentralized autonomous organization) for the contracts. In a DAO system, the DAO controls the contracts, and the users of the contracts control the DAO by using digital tokens which give them voting rights. The users who hold tokens (voting rights) can submit a proposal, e.g., executing the *selfdestruct* function. Then, the proposal will be checked by a group of volunteers called "curators" to check the legality of the submitted proposal and the identity of the submitter. Finally, the users who owned the DAO tokens vote to accept or reject the proposal.

However, adding a DAO pattern to the contract can increase code complexity, which is also a big concern according to our developer survey feedback. With the increase of code complexity, the probability of containing security vulnerabilities is also increasing, which leads to code related trust concerns for developers. To address these two concerns, we suggest that smart contract developers can open source and modularize this part of their code (DAO patterns) to a library. Other developers can then help polish the code together, and can make the code easier to use in the future.

Suggestion 4. Delay Self-destruct Action: The Ethers sent to a self-destructed contract will be locked forever, which increases the risk of using *selfdestruct* function. As we described in Section 4.1, the contract owner might find it difficult to inform all the users in a short time after the contract self-destructed. In this case, some users might send Ethers to the self-destructed contract and this may lead to financial loss. To address this problem, we suggest that the contract can delay the self-destruct action and throw an event to inform the users that the contract will self-destructed in

the near future. On the one hand, delaying self-destruct action can give time for voting (Suggestion 3). On the other hand, it can provide time to inform users that the contract will be destructed.

Suggestion 5. Pause Functionality: The options in Suggestion 3 and 4 require time to implement. However, when a contract is being attacked, any delay might lead to enormous financial loss. In this case, pausing the functionality when performing the methods described in Suggestion 3 and 4 are important. OpenZeppelin provides a *Pausable* contract template [57, 58], which can be easily used through inheritance. The three functions and two modifiers are shown in Listing 4. The two modifiers, i.e., *whenNotPaused* and *whenPaused*, can be added to control its states. Specifically, *whenNotPaused* makes functions callable only when the contract is not paused, and *whenPaused* makes a function callable only when the contract is paused. The state of the contract is obtained by a boolean value named *_paused*, which can change its state by the function *_pause()* and *_unpause().*

```
1  modifier whenNotPaused() { require(!paused(), "Pausable: paused"); _;}
2  modifier whenPaused() { require(paused(), "Pausable: not paused"); _;}
3  function paused() public view virtual returns (bool) { return _paused;}
4  function _pause() internal virtual whenNotPaused { _paused = true;}
5  function _unpause() internal virtual whenPaused { _paused = false;}
```

Listing 4. OpenZeppelin Pausable contract

Suggestion 6. Refund Values: Some smart contracts might store the values of users. For example, users might hold tokens in ERC20 contracts. Thus, destructing the smart contracts will lead to financial loss of users. Before executing the *selfdestruct* function, the contract owners should refund users' assets.

6.3 Tokens of Destructed Unmatched ERC20 Contracts

Unlike other kinds of contracts, ERC20 contracts usually store values (tokens) of users. Destructing the contract and transferring the balance is not enough for ERC20 contracts, as the tokens will be locked with the execution of the *selfdestruct* function. We investigate the transaction of 70 destructed unmatched ERC20 tokens introduced in Section 5.3 and find that 53 of them contain transactions that call *transfer()*, which means there are token transfers in these contracts. However, only 3 contracts have refund-related transactions. These three contracts pay back a certain amount of Ethers according to the tokens that users have. For the other 50 smart contracts, the contract owners destruct the contracts directly without considering the benefits of users. This situation might suggest that ERC20 tokens with *selfdestruct* function might have a high risk for users.

6.4 Inconsistency

6.4.1 selfdestruct functions on Etherscan vs. Survey Results. In this paper, we collected 54, 739 open-source smart contracts but only found 5.1% of them contain the *selfdestruct* function. However, in our survey, 38.82% of the developers claim that they will add *selfdestruct* functions to their contracts. There is an inconsistency between the practitioner's perception and their behavior (38.82% vs. 5.1%). The inconsistency might indicate that many developers admit the importance of *selfdestruct* function but they give up on adding it during the actual development process. The reasons why developers do not add the *selfdestruct* functions in actual developing process might have already been included in Section 3.3.2. Therefore, designing guidelines for using the *selfdestruct* function might be helpful. Future work can aim to design guidelines, development models or tools to address these problems. We also give five suggestions in the Section 6.2.

6.4.2 Self-destructed Contracts on Blockchain vs. Etherscan. We summarized 5 reasons why smart contracts were destructed based on 756 self-destructed contracts collected from Etherscan. However,

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:29

Table 6. self-destructed-related Info. on Ethereum blockchain

	Trace	self-destructed Contracts	Dest Accounts	Trans.
Total	30,486,241	2,084,841	19,131,801	374,735
Block No. 2.3M - 2.7M	27,560,501	4,522	19,127,397	53,374
Others	2,925,740	2,080,319	4,404	321,361

there are millions of destructed contracts on Ethereum blockchain. The inconsistency between the number of self-destructed Contracts on Ethereum blockchain and Etherscan might make our finding not so reliable. To investigate this inconsistency, we collected all self-destructed-related traces by Jan. 2019 (The date is the same as the verified contract we collected from Etherscan.) Table 6 shows the self-destructed-related information we collected from Ethereum blockchain. There are 30,486,241 self-destructed traces which were generated by 2,084,841 self-destructed contracts. When a contract is destructed, the contract will transfer its balance to another account; we call these contracts used to receive the balance as "dest" accounts. We found 19,131,801 dest accounts. Noticed that one transaction can have several self-destructed-related traces and one self-destructed-related trace related to one self-destructed operation. There is a difference between the number of self-destruct traces and the number of self-destructed contracts because a contract that has been marked as *destructed* still exists until the end of the transaction. The contracts can still be called and may execute further self-destructs. This mechanism was used by attackers to launch the DDoS attack in block No. 2.3M to 2.7M [44]. 27,560,501 (90.4%) self-destruct traces and 19,127,397 (99.98%) dest accounts were generated in this DDoS attack. Thus, to make the analysis more accurate, we removed the self-destructed-related information. generated on this DDoS attack, and there are 2,925,740 self-destruct traces generated by 2,080,319 self-destructed contracts. An interesting finding is that all the 2,080,319 self-destructed contracts are related to only 4,404 dest accounts. Among these 4,404 dest accounts, 2,716 are EOAs, 1,263 are self-destructed contracts and 425 are alived contracts (contracts that can be found on blockchain), while the number of dest accounts of 756 destructed contracts collected from Etherscan is 472 (415 EOA, 16 self-destructed contracts and 41 alived contracts). It shows that although there are large number of self-destructed contracts on Ethereum blockchain, most of them were generated by a limited number of accounts.

Previous work [34] also investigated the self-destructed contracts on Ethereum. They analyzed 7.3 M self-destructed contracts, and 7.2M (98.6%) were short-lived contracts (4.2M, 57.53%) [8], GasTokens (2.8M, 38.35%), and ENS (Ethereum Name Service) deeds (0.2M, 2.74%). All these 98.6% self-destructed contracts will not be included in the analysis of these papers because these contracts usually do not contain much information. For example, the recommended template of recommended GasToken [15] only contains seven instructions with two lines of code. Short-lived contracts are usually contract-created simple contracts that are created and destructed in a single transaction. Thus, there is no need to open source these contracts on Etherscan and will not be included in our analysis.

Our analysis and Di Angelo and Salzer's [34] finding can show that the self-destructed contracts we collected from Etherscan have a good coverage of selfdestruct function usages apart from DDoS attack, GasToken, and ENS deeds. It is also likely that we might miss to report some usages as the limitation of manual analysis used in this paper. However, this paper also highlights a new direction and might trigger more studies in the future.

6.4.3 GasToken Contracts on Blockchain vs. Survey Results. Previous work [34] reported that about 38.35% self-destructed contracts on Ethereum blockchain are GasToken. In our survey, only one

---

[8]A contract called short-lived when the creation and subsequent selfdestruct is executed in the same transaction.

Table 7. Open Interview Questions (Excerpt)

ID	Question
1	Do you used selfdestruct function?
2	Do you know the gas refund feature?
3	Do you know selfdestruct function can refund gas?
4	Do you add selfdestruct function for refunding gas? why?
5	Do you know GasToken and how do you think about it?

respondent mentioned that the motivation of including the *selfdestruct* function is refunding gas. To investigate this inconsistency, we performed an interview to collect developers' perspectives. From our survey, 33 respondents claim that they will add the selfdestruct function in their smart contracts. 18 out of 33 left their emails, and we sent emails to ask them whether they agreed to have a further interview about the gas refund feature. Finally, 8 respondents accepted the interview invitations, and all interviews were performed remotely via Skype or WeChat. We conducted semi-structured interviews followed by Zhou et al.'s method [70]. Specifically, we first introduced our work and asked the demographic questions shown in Section 3.2.2. The interviewees had an average experience of 3.9 years in smart contract developments with various roles, including 3 developers, 1 manager, 1 tester and 2 researchers. Then, we used some open questions listed in Table 7 to guide the discussion. The first question is used to confirm their qualification and all interviewees answered "*Yes*". For the second question, 6 interviewees said they did not know it, and the interviews were finished. For the other 2 interviewees, both said they had a deep understanding of the gas refund feature and knew the selfdestruct function could refund gas. Thus, we continued to the fourth and fifth questions to collect their feedback. As our interviews were semi-structured, we also asked follow-up questions to dig deeper according to their answers. For example, one interviewee said they have no need to use selfdestruct to return gas back, and we then asked them why this was the case.

We found the following reasons why they will not include *selfdestruct* function just for refunding gas. First, the most important reason is the lack of knowledge about this feature. Although the interviewees had an average experience of 3.9 years in smart contract developments, only two of them know this feature. Second, one interviewee said they focus more on the functionalities and the security of smart contracts. They do not pay for the gas and thus they do not care how to save gas for the contracts. Besides, both two interviewees mentioned that adding *selfdestruct* function will increase development cost. However, only up to half of the gas used by a transaction that calls the *selfdestruct* function will be returned. They believe the loss outweighs the gain.

In terms of the GasToken, the two interviewees gave various views, but all the views were negative. The first interviewee mentioned that he did not believe GasToken can save gas or used it to make profits. Developers can only save a little gas fee when the gas price on free time is several times higher than mint time. If the gas price on free time is similar or lower than mint time, using GasTokens will even cost more, as the creation of the GasTokens will also cost gas. Besides, the miner can only receive half of the gas fee because of the GasToken, and they might refuse to process the transaction. Thus, the usage of GasTokens seems narrow. Another interviewee said that GasToken is harmful for the Ethereum ecosystem, as it will create a large number of useless contracts on Ethereum. All of these useless contracts will be stored in nodes because of the distributed ledger nature of Ethereum, which wastes many storage and network resources.

Except from the views we collected from the interview, in the previous subsection (Section 6.4.2), we found that all the 2,080,319 self-destructed contracts only related to 4,404 dest accounts. This also shows that although there are a large number of GasTokens on Ethereum, they are only used

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:31

by a small number of users. It is an interesting topic to conduct a comprehensive investigation of the GasTokens, e.g., their main usage scenarios, but they are out-of-scope to this paper. Thus, we will conduct the investigation in our future work.

6.5 Threats to Validity

Internal Validity. In RQ1, we sent our survey to 996 developers and received 88 responses. The response rate is 8.84%. We used the feedback of these 88 responses to summarize key reasons why developers include or exclude *selfdestruct* function in their contracts. Due to the limited number of feedback. There might however still be other reasons we did not cover in our survey. We collected all contributors emails from the top 100 most popular smart contract related projects. We also tried to make our survey as simple as possible and give 2 respondents $50 Amazon gift card to increase the response rate. We finally obtained a 8.84% response rate, which is also acceptable [68].

In RQ2, we use SMARTEMBED to compute the similarity between smart contracts. If the similarity of two contracts larger than 0.6, we think they are a predecessor contract and its successor contract. The similarity threshold can influence the manual effort we need to pay. If the similarity is too large, we might miss some predecessor contracts and their successor contract. Otherwise, if the similarity is too small, we need to pay more effort to distinguishing whether the two contracts are relevant or not. The similarity threshold used in the paper of SMARTEMBED is 0.95, and it found few contracts are relevant if the similarity is lower than 0.7. To reduce the number of unidentified relevant contracts, we conservatively reduce the threshold to 0.6. We used *Open Card Sorting* to find 5 common self-destruct reasons. Due to the limitation of our understanding of the smart contracts, we might miss some self-destruct reasons. To reduce the threat of human factors, we followed the process of card sorting strictly, and the developers all have rich experience (>3 years) in smart contract related research. Researchers can also use the same method we proposed in RQ2 to find other self-destruct reasons in the future.

In RQ3, we use contracts whose number of transactions are larger than 500, and balance is larger than 1 Ether as the ground truth. We regard these contracts having a low probability of having permission problems. However, it is still possible we might find some functions in this group that have permission problems, but we believe the number of these functions is small.

We performed an interview to collect why developers do not include selfdestruct function for refunding gas. However, only 8 respondents accepted our interview, and 6 of them did not provide much information as they do not know the gas refund feature of Ethereum. Thus, it is likely that some points might be missing. Fortunately, we still obtain many reasonable feedbacks which can answer the inconsistency. We acknowledge that a comprehensive and convincing investigation about the GasTokens is an interesting topic, but it is not the core focus of this paper and needs further research efforts that we reserve for future work.

External Validity. The smart contracts used in this paper were up to Jan. 2019. *Solidity*, the most popular programming language for the smart contract, is fast-growing. From Jan. 2019 to the time of writing, there are 11 versions updated and released. Many new features have been removed and added in these versions. Ethereum also might be updated in the future through a hard fork [19]. In this case, the self-destruct reasons might be changed because of a major update to Ethereum and Solidity. Addressing this threat needs more research effort, but the method we proposed to find the self-destruct reasons is still working.

In this paper, we first cluster contracts with their creator addresses. However, it is likely that some developers may use multiple addresses to deploy smart contracts. In this case, our method may fail in finding successor contracts of some self-destructed contracts, and our analysis may miss uncovering some reasons for the use of self-destruct. Because of the anonymity of Ethereum, it is difficult to cluster contracts with multiple creator addresses, even if they are owned by the same

developer. Fortunately, from the collection of 756 self-destructed contracts that we analyzed, we find that for 436 of them we could find their successor contracts. In total we found 1513 *<predecessor contract, successor contract>* pairs. We use open card sorting to summarize the reasons why contracts self-destructed (details see Section 4.2.3). There are two iterations in the open card sorting. In the first iteration, we use 20% of the cards (one card is a *<predecessor contract, successor contract>* pairs) and get all the reasons. No new reason was found by using the remaining 80% of cards in the second iteration, which means 1513 pairs were enough to support the analysis work reported in this paper.

7   RELATED WORK

SMARTEMBED [42, 43] is the first tool that uses a clone detection method to detect bugs in smart contracts. The tool contains a training phase and a prediction phase. In the training phase, their dataset contains two parts, i.e., source code database and bug database. The source code database consists of the source code of all the open source smart contracts in the Ethereum. The bug database records the bugs of each smart contract in a source code database. SMARTEMBED first converts each smart contract to an AST(abstract syntax tree). After normalizing the parameters and irrelevant information on the AST, SMARTEMBED transfers the tree structure to a sequence representation. Then, they use *Fasttext* [23] to transfer code to embedding matrices. Finally, they compute the similarity between the given smart contracts with contracts in their database to find the clone contracts and clone related bugs. Although the tool is aimed at finding bugs, their first step is computing the code similarity between the given smart contract and history contracts in their database. Therefore, we can modify their code to compute the similarity between two given smart contracts (used in RQ2).

Bartoletti et al. [22] found that the infamous Ponzi schemes migrated to the digital world. Many frauds use Ethereum to design Ponzi schemes contracts for earning money. They manually analyzed 1,382 verified smart contracts on Etherscan and find 137 of them are Ponzi scheme contracts. Then, they divided these Ponzi scheme contracts into four categories, i.e., array-based pyramid schemes, tree-based pyramid schemes, handover schemes, and waterfall schemes. Bartoletti et al. opened their dataset to the public but do not provide a tool to detect whether a contract is a Ponzi scheme contract. To address this limitation, Chen et al. [67] proposed a method that uses a machine learning algorithm (XGBoost [29]) to distinguish Ponzi scheme contracts. They use account features and code features to train the module. The account features are extracted from the transactions, e.g., the number of payment transactions, the balance in the contracts. The code features can be obtained from contract bytecode. They count the frequency of each opcode in the contract bytecode. Both account features and code features do not need the source code of contracts. Therefore, their method can predict arbitrary contracts on the Ethereum.

Oyente [52] is the first tool for security examination for smart contracts based on symbolic execution. Their work introduces four security issues on smart contracts, i.e., mishandled exception, transaction-ordering dependence, timestamps dependence, and re-entrancy attack. To detect these security issues, Oyente first constructs a CFG (control flow graph) based on symbolic execution. After that, they design different rules to detect these four security issues. Kalra et al. [48] proposed a tool named Zeus, which can detect seven kinds of security problems; four of them are the same with Oyente; the other three issues are *failed send, interger overflow/underflow* and *transaction state dependence*. Zeus can detect security issues at the source code level. They use LLVM bytecode to represent the Solidity source and detect related patterns through LLVM bytecode. Contract-Fuzzer [46] is the first fuzzer to detect seven security issues in smart contracts. Four security issues are the same as Oyente; the other three issues are *gasless send, dangerous delegatecall* and *freezing ether*. ContractFuzzer utilizes ABI (abstract binary interface) of smart contracts to generate fuzzing inputs and defines test oracles to detect security issues.

Chen et al. [26] define 20 smart contract defects on Ethereum. They first crawl 17,128 Stack Exchange posts and use key words to filter solidity related posts. After getting Solidity related posts, they use *Open Card Sorting* to find 20 contract defects and divide them into five categories, i.e., *security, availability, performance, maintainability*, and *reusability defects*. According to their paper, although previous works define several security defects, they did not consider the practitioners' perspective. Therefore, they design an online survey to collect feedback from developers. The feedback shows that all the defined contract defects are harmful to smart contracts. They assign five impact levels the defined 20 contract defects. Defects with impact level 1-3 can lead to unwanted behaviors of contract, e.g., crashing or a contract being attacked.

Li et al. [50] proposed a symbolic execution analysis tool named SOLAR to detect violations of two standards, i.e., ERC20 and ERC721. SOLAR is built on top of Manticore [53] which is a well-known symbolic execution framework for smart contracts, and uses boolector [55] as the SMT solver to check the symbolic constraints. Since Manticore does not fully support EVM instructions, SOLAR is extended to fully support EVM instructions. Their experimental results show that SOLAR is significantly more effective than previous tools, e.g, Mythril [5], and can find more errors with fewer false positives.

Di Angelo and Salzer. [34] introduced the usage of smart contracts on Ethereum by analyzing about 20 million deployed smart contracts. They defined ten kinds of usages of smart contracts on Ethereum and found that most of the deployed smart contracts remain unused and tokens are the most popular applications of Ethereum. self-destructed contracts is one important research dimension in their work. They found 7.3M self-destructed contracts on their dataset. According to their analysis, 4.2 M of them were created and then self-destructed in the same transaction; 2.8 M were GasTokens, and 0.2 M were ENS (Ethereum Name Service) deeds. Eight thousand of the remaining nine thousand contracts were self-destructed for unknown reasons and 778 were wallets. Their other work [33] also investigated the usage of self-destructed contracts and found 48,506 contracts self-destructed multiple times (some of them up to 10 920 times). They called a contract a mayfly if the contract was created and self-destructed in the same transaction. They found 1,856,655 mayflies that were created by just 8,992 distinct addresses, and most of the mayflies appeared during the DDos period of 2016. All of their work and this work investigated the usage of selfdestruct function on Ethereum, but our work is more comprehensive and has a different focus. Specifically, we first conducted an online survey to collect developers' feedback about why they add or do not add selfdestruct function. Then, we proposed a method to find the reasons why smart contracts self-destructed. Finally, we propose a tool to detect two problems that might shorten the lifespan of smart contracts.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we conducted a comprehensive empirical study on the use of the *selfdestruct* function on Ethereum. To understand the smart contract developers' perspective, in RQ1, we designed an online survey to collect reasons from developers why they include and exclude the *selfdestruct* function in their contracts. We summarized 6 reasons for including and 6 reasons for excluding *selfdestruct* function in their contracts, respectively. The feedback also shows that 22 / 33 respondents claim that they add *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. According to this information, we propose an approach that can find the upgrade version of the self-destructed contracts in RQ2. After that, we used the *open card sorting* method and summarized 5 reasons why contracts might destruct. Two of them – *Unmatched ERC20 Token* and *Limits of Permission* – can affect the life span of smart contracts. To detect these problems, we developed a new tool named LifeScope in RQ3, which reports 0 false positive / negative in detecting *Unmatched*

ERC20 Token, and achieves an F-measure and AUC of 77.89% and 0.8673 for detecting the *Limits of Permission* issue. Finally, to help developers use the *selfdestruct* function better, we give 6 suggestions based on the feedback of our survey and our smart contract analysis.

Apart from the *selfdestruct* function, using *Delegatecall* or *Callcode* can also be used to design an upgradeable smart contract [8]. Specifically, we need a proxy contract and a logic contract to design an upgradeable contract. The proxy contract stores the storage variables and Ethers, and the logic contract contains the logical code. The proxy contract uses *Delegatecall* to call the code of the logic contract. If there are some security issues found on the logic contract, developers can deploy a new logic contract and discard the old logic contract. By comparing the difference between the old logic contract and new logic contract, we can find some reasons why contracts need upgrade. In Section 6.3, we only investigate the transaction of destructed unmatched ERC20 tokens. However, it is likely that the matched ERC20 tokens also do not transfer values back to the users. Also, the upgradeable contracts will also lock the tokens of users. In the future, we will conduct more comprehensive work to investigate the value lock on ERC20 tokens.

We found 2,789 (5.1%) smart contracts in total contain the selfdestruct function, while only 199 (0.36%) contracts contain the Delegatecall or Callcode functions. The data size is one of the main reasons why we choose to investigate *selfdestruct* function first. In the future, we plan to conduct an empirical study to investigate the upgradeable smart contracts on Ethereum when there are more smart contracts that contain Delegatecall or Callcode function. Specifically, there are many new issues of designing an upgradeable smart contract. For example, although the old logic contract is discarded, the contract can still be called by other contracts, which might lead to new security issues; designing an upgradeable smart contract can also increase the development cost [27]. Thus, we first plan to investigate developers' motivation about why they design their contracts as upgradeable. Then, we plan to design a method to find the *<old logic contract, new logic contract>* pairs. Unlike *selfdestruct* function where we need to compare similarity to find the pair, for the upgradeable contract, we can check the transaction details of the proxy contract. From the transactions, we can find the discarded logic contract and new logic contract. After that, we can compare their differences to find the reasons why developers upgrade a smart contract. Finally, we will update LifeScope to detect the additional problems.

We analyzed self-destructed contracts with source code and conducted a preliminary analysis for the self-destructed contracts on Ethereum blockchain. In the future, we will perform a comprehensive empirical study to analyze the self-destructed contract on Ethereum. Specifically, we first investigate how many GasTokens on Ethereum; how many users of the GasTokens, and the usage scenarios of the GasTokens. Besides, it is interesting to analyze remaining self-destructed contracts. For example, are there any other DDoS attacks happening; why millions of self-destructed contracts are only related to 4,404 dest accounts, and why some developers created a large number of self-destructed contracts.

We observed the difference between a PS pair to summarize the reasons why smart contracts were destructed. However, we only stand at a high level to present our observations instead of digging out a more detailed reason. Specifically, *Setting Changes* and *Functionality Changes* are two broad definitions. Developers might change the functionalities for several reasons, e.g., adding business requirements, fixing bugs, increasing readabilities. Besides, when multiple changes happen, we regard all of them as the reasons that contributed to the self-destruct. Actually, their importance might be different, and maybe only one of them was the real reason why smart contracts were destructed. In the future, we will conduct a more comprehensive empirical study to find more specific reasons that lead a contract self-destruct.

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:35

REFERENCES

[1] Apr., 2018. *Understanding The DAO Attack.* https://www.coindesk.com/understanding-dao-hack-journalists/
[2] Apr., 2019. *Decentralized application.* https://en.wikipedia.org/wiki/Decentralized_application
[3] April., 2018. *ERC20.* https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
[4] Aug., 2019. *Ethereum Smart Contract Security Best Practices.* https://consensys.github.io/smart-contract-best-practices/
[5] Aug., 2019. *Mythril: Security analysis tool for EVM bytecode.* https://github.com/ConsenSys/mythril
[6] Aug., 2019. *Ripple Coin.* https://ripple.com/
[7] Aug., 2020. *DiffChecker.* https://github.com/trembacz/diff-checker
[8] Dec., 2020. *EIP-2535: Diamond Standard.* https://eips.ethereum.org/EIPS/eip-2535
[9] Feb., 2019. *Cryptokitties.* https://www.cryptokitties.co/
[10] Feb., 2019. *Ethereum Classic.* https://en.wikipedia.org/wiki/Ethereum_Classic
[11] Feb., 2020. *Expanse.* https://coinswitch.co/info/expanse/what-is-expanse
[12] Jan., 2019. *Blockchain.* https://en.wikipedia.org/wiki/Blockchain
[13] Jan., 2019. *Cryptocurrency.* https://en.wikipedia.org/wiki/Cryptocurrency
[14] Jan., 2019. *Ethereum.org.* https://www.ethereum.org/
[15] June., 2021. *GasToken.io.* https://gastoken.io/
[16] Mar., 2018. *EtherScan.* https://etherscan.io/
[17] Mar., 2018. *The Solidity Contract-Oriented Programming Language.* https://github.com/ethereum/solidity
[18] Mar., 2018. *Solidity Document.* http://solidity.readthedocs.io
[19] Oct., 2019. *Blockchain Hard Fork.* https://www.investopedia.com/terms/h/hard-fork.asp
[20] Sept., 2019. *Camel Case.* https://en.wikipedia.org/wiki/Camel_case/
[21] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern information retrieval.* Vol. 463. ACM press New York.
[22] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2020. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *Future Generation Computer Systems* 102 (2020), 259–277.
[23] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[24] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. 2019. Understanding the motivations, challenges and needs of blockchain software developers: A survey. *Empirical Software Engineering* 24, 4 (2019), 2636–2673.
[25] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Amiangshu Bosu. 2018. Understanding the software development practices of blockchain projects: a survey. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10.
[26] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* (2020).
[27] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2020. Maintaining Smart Contracts on Ethereum: Issues, Techniques, and Future Challenges. *arXiv preprint arXiv:2007.00286* (2020).
[28] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proceedings of the 27th Network and Distributed System Security Symposium.*
[29] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining.* ACM, 785–794.
[30] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 1503–1520.
[31] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. 2018. Understanding ethereum via graph analysis. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications.* IEEE, 1484–1492.
[32] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[33] Monika Di Angelo and Gernot Salzer. 2019. Mayflies, breeders, and busy bees in Ethereum: smart contracts over time. In *Proceedings of the Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts*. 1–10.

[34] Monika Di Angelo and Gernot Salzer. 2020. Characterizing types of smart contracts in the ethereum landscape. In *International Conference on Financial Cryptography and Data Security*. Springer, 389–404.

[35] Dmitry Efanov and Pavel Roschin. 2018. The all-pervasiveness of the blockchain technology. *Procedia Computer Science* 123 (2018), 116–121.

[36] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. 2019. The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* (2019).

[37] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE transactions on software engineering* (2018).

[38] Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. 2019. ÆGIS: Smart Shielding of Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2589–2591.

[39] Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. 2015. Longest common substrings with k mismatches. *Inform. Process. Lett.* 115, 6-8 (2015), 643–647.

[40] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[41] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. 2019. Detecting token systems on ethereum. In *International conference on financial cryptography and data security*. Springer, 93–112.

[42] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding. *35th IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2019).

[43] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking Smart Contracts with Structural Code Embedding. *IEEE Transactions on Software Engineering* (2020).

[44] Richard Greene and Michael N Johnstone. 2018. An investigation into a denial of service attack on an ethereum network. (2018).

[45] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[46] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[47] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[48] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[49] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to advanced empirical software engineering*. Springer, 63–92.

[50] Ao Li and Fan Long. 2018. Detecting standard violation errors in smart contracts. *arXiv preprint arXiv:1812.07702* (2018).

[51] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2017. A survey on the security of blockchain systems. *Future Generation Computer Systems* (2017).

[52] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[53] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[54] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[55] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation* 9, 1 (2014), 53–58.

[56] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

[57] Openzeppelin. June., 2021. *Pausable Documentation.* https://docs.openzeppelin.com/contracts/2.x/api/lifecycle

[58] Openzeppelin. June., 2021. *Pausable Documentation.* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/b0cf6fbb7a70f31527f36579ad644e1cf12fdf4e/contracts/security/Pausable.sol

[59] Martin Porter. 1980. An algorithm for suffix stripping. Program, 3 (14): 130–137.

[60] Fabrizio Sebastiani. 2002. Machine learning in automated text categorization. *ACM computing surveys (CSUR)* 34, 1 (2002), 1–47.

[61] Donna Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.

[62] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and*

Why Do Smart Contracts Self-Destruct?
Investigating the Selfdestruct Function on Ethereum

1:37

    Communications Security. ACM, 67–82.

[63] Pradeep K Tyagi. 1989. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science* 17, 3 (1989), 235–241.

[64] Buterin Vitalik and Swende Martin. June., 2021. *EIP-3298: Removal of refunds*. https://eips.ethereum.org/EIPS/eip-3298

[65] Buterin Vitalik and Swende Martin. June., 2021. *EIP-3529: Reduction in refunds*. https://eips.ethereum.org/EIPS/eip-3529

[66] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. Vultron: catching vulnerable smart contracts once and for all. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 1–4.

[67] Jiahui Cui Edith Ngai Peilin Zheng Weili Chen, Zibin Zheng and Yuren Zhou. 2018. Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1409–1418.

[68] Xin Xia, Zhiyuan Wan, Pavneet Singh Kochhar, and David Lo. 2019. How practitioners perceive coding proficiency. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 924–935.

[69] Yiming Yang and Jan O Pedersen. 1997. A comparative study on feature selection in text categorization. In *Icml*, Vol. 97. Nashville, TN, USA, 35.

[70] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019).